

JSP 2.0 技术手册

林上杰 林康司 编著

电子工业出版社

Publishing House of Electronics Industry

北京 • BEIJING

推荐序

认识林上杰和林康司大概是三年前的事情了。

林康司是一个个性温和害羞的大男生。有他共同执笔这本书，可以让这本书的步调慢一点，优雅一点。

我认识的林上杰则是一个极端龟毛（注：台湾口语，挑剔的意思）的人。

记得第一次找他帮 Sun 教育训练中心上课，对象是业界具有“员工平均水准最高”之称的公司。虽然课后问卷的结果令人满意，可是他还是深深地自责，觉得自己没有教好。

当然，现在的他又比当时更进步了。把对自己的高标准用在写作上，这当然是一本高标准的书籍。从写作的题材、校稿、排版到封面设计，无一不是他绞尽脑汁的结果。对我这种只负责把稿子交给出版社的作者来说，实在是强烈的对比，也难怪我曾被他嘲笑，说我的书除了著作人的名字看起来有 60 分之外，其他的全部不及格。说实话，我不能同意他更多了。

这本书的重点是介绍 Java 在展示层的两项重要技术：Java Servlet 与 JavaServer Pages。它们是最重要的 Java 核心技术，对这两项技术有深入的了解，将有助于您未来对于 JavaServer Faces(JSF) 技术，以及 Java Web Services 技术的学习。侯捷老师常常告诫大家：“勿在浮沙筑高台”。即使 JSF 和 Java Web Services 技术将底层包得很漂亮，工程师也不必去接触底层的实现细节，但是对基础技术的了解，只会让我们在使用技术时感觉更实在，运用得更漂亮。有两位作者对品质的高度要求，相信这一定是一本可以带大家入门 Java 展示层技术的优秀书籍。在此诚为各位推荐。

王 森
Sun Microsystems
教育训练中心经理
2004 年 3 月

JSP2.0 技术手册

序 —

记得三年前

亲自带着刚上市的《JSP 技术手册》送给我最敬重的教授，从他的表情中，可以看出他十分地高兴、满意，在一段不算短的谈话中，他一直不断地重复提到：

写书是一件具有影响力的事情，一本好书可以影响很多人，而不好的书更可能误了许多人。

感谢昔日恩师的这段话在我的心中埋下种子。今日，我正怀着戒慎恐惧的心情，完成此书。

感谢：

本书能够顺利完成，必要感谢许许多多的人。首先感谢王森(moliwang)先生的协助，假若此书遭到众人的讨伐，王森先生亦将受池鱼之殃。感谢松凌科技李日贵(jini)先生在我撰写本书时，提供题材选取的建议和技术问题的解答。感谢吴卢基(Worookie)先生校阅本书的内容，修改许多大大小小的错误，让本书的品质能够更上一层楼。

感谢小傅子(Dennyfu)协助本书的封面设计，和他合作封面设计，乃写书最大之乐趣。感谢许芳凰(Mina)小姐协助本书的图片设计，让书中的图片不再是丑陋的线条。

感谢博文视点(Broadview)资讯有限公司的编辑周筠女士及方舟先生，有了他们的协助，才使得本书的简体版能顺利出版。

感谢我的最爱王斐珊，有你的扶持，使我能够顺利度过这段寂寞又难熬的日子。感谢你在除夕夜至大年初三，协助我润稿。希望此书能够大卖，让我能早日累积起我俩的结婚基金。

最后要感谢我的家人持续不断地支持我，让此书能在最短的时间内完成。

一本书所能介绍的内容有限，为了弥补此项缺憾，特架设 Java 技术论坛：<http://www.javaworld.com.tw>，希望书中缺少的内容，读者都能在论坛中找到解答。

林上杰
2004 年 3 月

JSP2.0 技术手册

序 二

回想在大学时江虹庆先生和林上杰先生带我认识 Java，至今已三年多了。尤其在当时 Servlet/JSP 相关的资源还非常的少，对从完全不懂 Java 且一入门就从 Servlet 开始的我而言，他们的耐心和认真的指导帮我开启了 Java 的一扇窗。

这次能把从许多前辈那里学习来的东西(不管是周遭或者是网络上的朋友)，写成书回馈给更多读者，令我感到非常高兴。

跟当时的环境比起来，现在想进入这个领域的人真的很幸福。在 Google 打上关键字就有看不完的信息，在 JSPTw.com (www.javaworld.com.tw 的前身)Java 技术论坛中，更可以搜寻到许多相关的精华文章，书架上又有好几本不同作者的书可供读者选择，尤其是你手中的这本书©这本书结合了林上杰先生与我的心血，由浅入深，作为入门书，本书无疑是最佳选择；对于进阶者来说，本书更提供了 JSP 2.0 新增功能的使用说明。希望大家看完这本书后，能对 Java 在 Server-side 上的应用产生兴趣，进而接触得更多、更深、更广。

再次感谢林上杰先生让我有机会参与本书的撰写，能与他共事是我的一大荣幸，更感谢 JSPTw 版友的知识分享与讨论，以及容忍我花时间在书上的家人和我的女友新玫，也谢谢花时间正在看此序的您。

林康司
2004 年 3 月

JSP2.0 技术手册

导 读

本书内容分成三大部分。

前三章主要内容为：从安装设定执行环境、范例程序到简介 Servlet 2.4 跟 JSP 2.0。

阅读完这三章之后，可以让读者奠定相关的概念基础，并且清楚了解 Servlet 2.4 和 JSP 2.0 上的进展。

第四章到第十一章为学习 JSP 2.0 的基础章节，从 JSP 2.0 基本语法、隐含对象、Expression Language 和 JSTL 1.1，一直到 Session Tracking 和 Filter 与 Listener 的介绍。这几个章节皆以笔者认为合适的顺序循序介绍，希望能够让读者在阅读本书后面部分的时候不会感到突兀。

介绍完基本概念和基础课程后，笔者在这里安插一个章节来介绍 JSP 执行环境与开发工具。读者可以在此章学习到最新 Tomcat 5.0 的基本使用，当然笔者没有漏掉目前在 Java IDE 界掀起一片旋风的 Eclipse。通过本章的介绍，各位读者会了解到如何使用 Eclipse 来开发 Web Application。

完成前两大部分的学习后，接下来就是第十三章到第十八章的提高部分。第十三章为第十四章的前置章节，因为在学习 JSP 与 JDBC 之前必须对 SQL 有相当的认识。第十四章的 JSP 与 JDBC，不只介绍基本的 JDBC 使用，也进而介绍好用的 Connection Pool API 的 Proxool。

第十五章和第十六章分别介绍 JSP 2.0 自定义标签的做法。第十五章简单介绍以往开发标签的方式。第十六章介绍在 JSP 2.0 新增开发标签的方式：Simple Tag 与 Tag File，通过几个简单的范例演练，轻松学会使用 Simple Tag 与 Tag File。

第十七章介绍其他的相关应用：JavaMail。第十七章除了介绍如何使用 JavaMail 来传送信件之外，还说明了如何传送附件和 HTML 格式的信件。

第十八章则是探讨 Web 应用程序设计师应有的设计理念和实现方式，使读者掌握开发高弹性、易维护的 Web 应用程序所应有的观念。

最后本书的附录 A：安装 Linux 执行环境。常常会有人提到 Java 时就会想到 Linux，虽然这两者基本上没有任何关系，但是在建构稳定的 Web Application 时，Linux 确实是一个不可错过的平台。因此，笔者把原本在 win 平台上的安装设定，重新编辑成 Linux 平台版本，供读者参考。

另外，您不能错过附录 B，在本附录所列出的 Servlet 2.4 / JSP 2.0 的 web.xml 中有一些设定好的项目，那是一份有用的参考内容。

相信通过循序渐进的学习，读者定能驾轻就熟，游刃有余。

JSP2.0 技术手册

目 录

第一章 安装执行环境	(1)
1-1 安装 J2SDK 1.4.2	(2)
1-2 安装 Tomcat 5.0.16	(6)
1-3 安装 JSPBook 站台范例	(11)
1-4 安装 Ant 1.6	(13)
第二章 Servlet 2.4 简介	(17)
2-1 Servlet 简介	(18)
2-2 First Servlet Sample Code	(19)
2-3 Servlet 的生命周期	(21)
2-4 Servlet 范例程序	(23)
2-5 Servlet 2.4 的新功能	(25)
第三章 JSP 2.0 简介	(31)
3-1 JavaServer Pages 技术	(32)
3-2 What is JSP	(33)
3-3 JSP 与 Servlet 的比较	(33)
3-4 JSP 的执行过程	(34)
3-5 JSP 与 ASP 和 ASP+ 的比较	(40)
3-6 JSP 2.0 新功能	(43)
第四章 JSP 语法	(47)
4-1 Elements 和 Template Data	(48)
4-2 批注 (Comments)	(48)
4-3 Quoting 和 Escape 规则	(49)
4-4 Directives Elements	(52)
4-5 Scripting Elements	(59)
4-6 Action Elements	(61)

JSP2.0 技术手册

4-7 错误处理	(68)
第五章 隐含对象 (Implicit Object)	(73)
5-1 属性(Attribute)与范围(Scope)	(75)
5-2 与 Servlet 有关的隐含对象	(81)
5-3 与 Input / Output 有关的隐含对象	(83)
5-4 与 Context 有关的隐含对象	(90)
5-5 与 Error 有关的隐含对象	(97)
第六章 Expression Language	(99)
6-1 EL 简介	(100)
6-2 EL 语法	(100)
6-3 EL 隐含对象	(104)
6-4 EL 算术运算符	(111)
6-5 EL 关系运算符	(113)
6-6 EL 逻辑运算符	(115)
6-7 EL 其他运算符	(116)
6-8 EL Functions	(118)
第七章 JSTL 1.1	(125)
7-1 JSTL 1.1 简介	(126)
7-2 核心标签库 (Core tag library)	(130)
7-3 I18N 格式标签库 (I18N-capable formatting tags library)	(160)
7-4 SQL 标签库 (SQL tag library)	(180)
7-5 XML 标签库 (XML tag library)	(189)
7-6 函数标签库 (Functions tag library)	(201)
第八章 JSP 与 JavaBean	(219)
8-1 JavaBean 的简介	(220)
8-2 JSP 与 JavaBean	(222)
8-3 JavaBean 的范围	(234)
8-4 JavaBean 的移除	(239)
第九章 网页窗体的处理	(243)
9-1 HTML 窗体如何传送数据	(244)
9-2 窗体中常见的输入类型	(244)
9-3 JSP 处理窗体	(247)

9-4	文件上传——Oreilly 上传组件	(249)
9-5	jspSmartUpload——上传和下载	(260)
9-6	本文区输入类型 (Textarea)	(270)
第十章	Session Tracking	(275)
10-1	Stateful & Stateless	(276)
10-2	Session Tracking 的四种方法	(276)
10-3	Session 的生命周期	(282)
10-4	HttpSessionBindingListener 接口	(284)
10-5	Shopping Cart 范例程序一	(289)
10-6	Shopping Cart 范例程序二	(295)
第十一章	Filter 与 Listener	(307)
11-1	Filter 简介	(308)
11-2	Filter 的运作方式	(308)
11-3	实现阶段第一个 Filter	(311)
11-4	对请求做统一的认证处理	(314)
11-5	ServletRequest 和 ServletResponse 之 Wrapper 类	(320)
11-6	使用 Filter 来解决中文问题	(329)
11-7	Listener 接口简介	(331)
11-8	ServletContext Listener	(333)
11-9	HttpSession Listener	(337)
11-10	ServletRequest Listener	(341)
第十二章	JSP 执行环境与开发工具	(345)
12-1	Tomcat 5.0 的介绍	(346)
12-2	JSP 开发工具介绍	(350)
12-3	Eclipse 简介与安装	(350)
12-4	使用 Eclipse 开发 Hello World	(353)
12-5	使用 Eclipse 开发 Web Application	(357)
12-6	使用 Eclipse 来开发 Web Application(2)	(363)
第十三章	SQL 介绍	(369)
13-1	数据库基本概念	(370)
13-2	SQL 简介	(370)
13-3	DDL 语句	(382)

13-4	数据的查询 —— SELECT	(385)
13-5	新增数据—— INSERT	(396)
13-6	修改数据——UPDATE	(399)
13-7	删除数据——DELETE	(399)
第十四章	JSP 与 JDBC	(401)
14-1	JDBC 简介	(402)
14-2	MySQL 的安装与使用	(402)
14-3	JDBC 连接 MySQL	(411)
14-4	JDBC 连接 MySQL 的中文问题	(422)
14-5	PreparedStatement	(428)
14-6	CallableStatement	(430)
14-7	JDBC 2.0 介绍与使用	(432)
14-8	JNDI - 数据来源(Data Source)与连接池(Connection Pool)	(438)
14-9	JSTL 的 SQL 标签库	(442)
14-10	Connection Pool - Proxool	(445)
第十五章	JSP Tag Library	(455)
15-1	JSP Tag Library 简介	(456)
15-2	一个简单的 Tag Library 范例	(457)
15-3	Tag Handler Class	(462)
15-4	Tag Library 范例程序	(475)
第十六章	Simple Tag 与 Tag File	(487)
16-1	Simple Tag	(488)
16-3	Tag File	(497)
16-4	Tag Library Descriptor (TLD)	(508)
第十七章	JSP 与 JavaMail	(519)
17-1	JavaMail 1.3.1 的介绍与使用方法	(520)
17-2	JavaMail 范例程序一——传送一般邮件	(521)
17-3	JavaMail 范例程序二——传送 HTML 格式的邮件	(524)
17-4	JavaMail 范例程序三——传送附件	(527)
17-5	JavaMail 范例程序四——传送自定义内容的邮件	(531)
第十八章	Web 架构——MVC	(537)
18-1	MVC (Model - View - Controller)的介绍	(538)

18-2 Model 1 与 Model 2 的介绍.....	(539)
18-3 Model 1 和 Model 2 的范例程序.....	(542)
附录 A 安装 Linux 执行环境.....	(557)
A-1 安装 J2SDK 1.4.2.....	(558)
A-2 安装 Tomcat 5.0.16.....	(560)
A-3 安装 JSPBook 站台范例.....	(561)
A-4 安装 Ant 1.6.....	(563)
A-5 安装 Apache 2.0.48 + Tomcat 5.0.16.....	(565)
附录 B web.xml 元素介绍.....	(571)
附录 C 使用 JDBC-ODBC 桥接器连接 Access.....	(581)
附录 D JSP 资源.....	(585)
附录 E HTTP 状态码.....	(587)
附录 F ASCII 码.....	(591)
附录 G Apache License 1.1.....	(593)

1

第一章

安装执行环境

本章将以 Step by Step 的方式，说明在 Windows 下，如何安装 Servlet/JSP 执行环境和使用本书的范例程序。本章将分 4 节来介绍：

- 1-1 安装 J2SDK 1.4.2
- 1-2 安装 Tomcat 5.0.16
- 1-3 安装 JSPBook 站台范例
- 1-4 安装 Ant 1.6

JSP2.0 技术手册

1-1 安装 J2SDK 1.4.2

只要我们需要执行、开发任何有关 Java 程序时,首先都须先安装 Java 2 Software Development Kit, 简称 J2SDK 或者又称 JDK(Java Development Kit), 它主要包含:

- Java API
- Java Compiler “javac”
- Java Debugger
- Java Plug-in
- Java HotSpot Client Virtual Machine
- Java 2 RE(Java 2 Runtime Environment)

1-1-1 Java Compiler 和 Java Debugger

Java Compiler 和 Java Debugger 主要用来开发 Java 程序, 因此, Java 2 SDK 也可以算是最简易的 Java 程序开发工具。

1-1-2 Java Plug-in

Java Plug-in 主要让浏览器如 Internet Explorer 或是 Netscape 用来执行 Applet 的软件。

1-1-3 Java HotSpot Client Virtual Machine

Java HotSpot Client Virtual Machine 就是俗称的 JVM, 据 JavaSoft 官方网站宣称: Java 2 SDK 1.3 加入 HotSpot 是为了增进 JVM 的执行效率; 他们甚至说, 某些情况下 Java 的编译、执行时间会和 C++ 并驾齐驱, 并且它的效能比之前 1.0.1 版要快上 30%。但是, 老实说 Java 最令人诟病的问题还是在于执行效率太差, 希望它未来在效能上能够有更突出的表现。

1-1-4 Java 2 RE(Java 2 Runtime Environment)

简单地讲, JRE(Java Runtime Environment)就是一个标准 Java 应用程序的执行环境。JavaSoft 官方网站中, 也有单独将这一部分让用户方便下载。原因在于, 假若用户执行 Java 所写的应用软件或是 Applet 时, 你只要先安装 JRE 即可, 而不须再去下载一个庞大的 Java 2 SDK 来使用, 除非你本身也想要从事 Java 程序的开发, 才需要一套集成开发和执行环境的软件。

Tomcat 5.0.16 运作时, 必须要有 J2SDK 1.3.1 或以上的版本, 而 J2SDK 可以从 <http://www.javasoft.com> 网站自行免费下载, 其中 J2SDK 提供有 Solaris、Linux、Windows 三种平台的版本。目前 Windows 平台的最新 J2SDK 版本为 J2SDK 1.4.2 (见图 1-1)。或直接使用本书 CD 光盘中的 J2SDK 1.4.2, 软件名称为: *j2sdk-1_4_2_03-windows-i586-p.exe*。

JSP2.0 技术手册

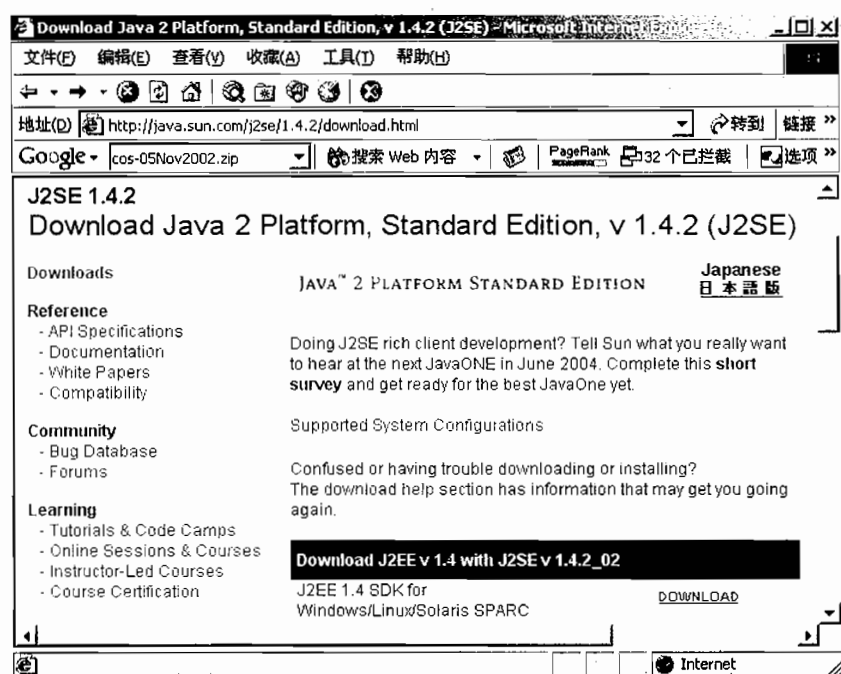
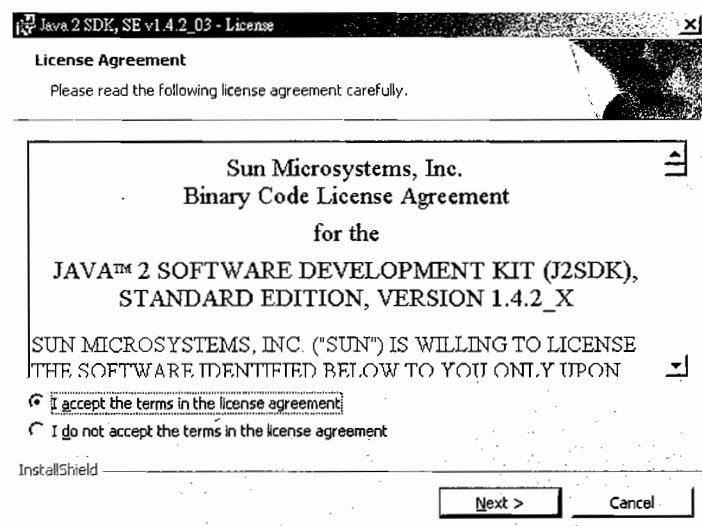


图 1-1 下载 J2SDK 1.4.2

第一步：执行 `j2sdk-1_4_2_03-windows-i586-p.exe`（见图 1-2）；

图 1-2 执行 `j2sdk-1_4_2_03-windows-i586-p.exe`

先按【Next】，选择【I accept the terms in the license agreement】后，再按【Next】。

第二步：选择安装路径及安装内容（见图 1-3）；

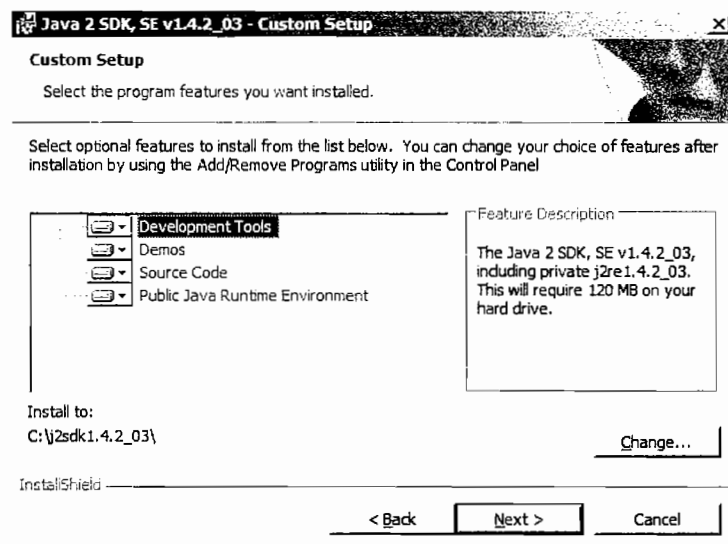


图 1-3 选择安装路径及安装内容

一般来说，我们通常都默认安装在 `C:\j2sdk1.4.2_03` 的目录下，假若你要安装在其他路径时，请按【Change】，改变 J2SDK 安装路径。确认无误后，再按【Next】。

随后开始安装 Java Plug-in 至浏览器上，一般都选择【Microsoft Internet Explorer】。再按下【Install】，正式开始执行安装程序，假若安装成功后，你会看到如图 1-4。

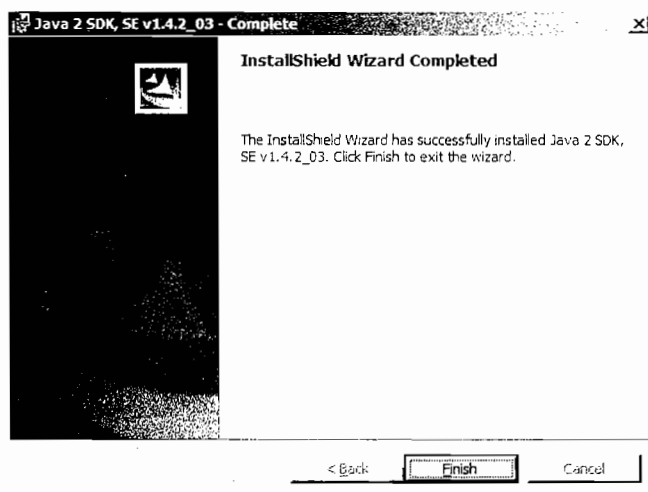


图 1-4 成功安装 J2SDK 1.4.2_03

JSP2.0 技术手册

第三步：设定 J2SDK 1.4.2_03（见图 1-5）：

从【开始】→【设置】→【控制面板】→【系统】→【高级】→【环境变量】→【系统变量】，然后到【新建】。

JAVA_HOME = C:\j2sdk1.4.2_03

PATH = %JAVA_HOME%\bin

CLASSPATH = C:\j2sdk1.4.2_03\lib\tools.jar;C:\j2sdk1.4.2_03\lib\dt.jar;

注意

1. CLASSPATH 的设定中，分号(;)用来分开两路径，切勿任意空格；
2. CLASSPATH 的设定中，分号的最后还有一个点“.”。

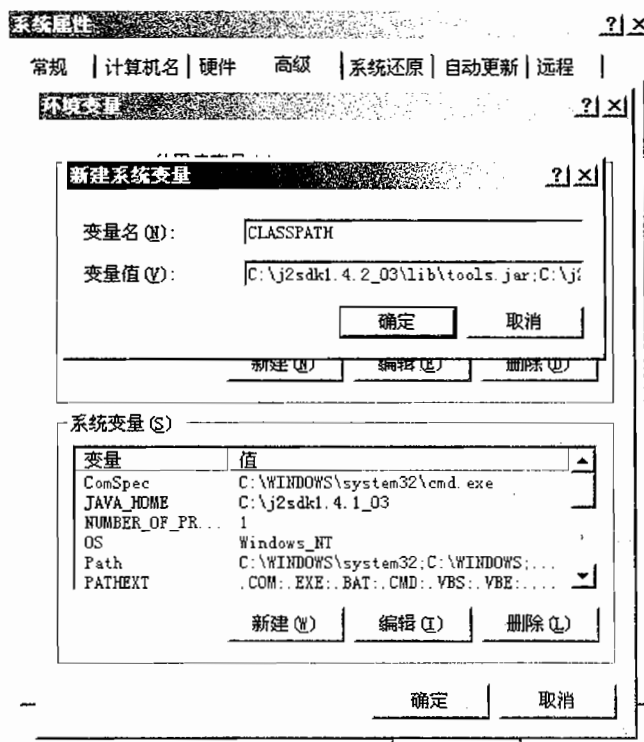


图 1-5 设定 J2SDK 之 CLASSPATH

补充

不论 Windows 2000 或 Windows XP 皆可依上述方法设定。

第四步：测试 J2SDK。

撰写一个 *HelloWorld.java* 程序，放置在 *C:\HelloWorld.java* 中。

■ *HelloWorld.java*

```
public class HelloWorld {  
  
    public static void main(String[] args) {  
        System.out.println("Hello World");  
    }  
}
```

打开命令提示符，在 *C:* 下输入 *javac HelloWorld.java*，然后再输入 *java HelloWorld*，执行 *HelloWorld* 程序，假若顺利成功，则会显示“Hello World”，如图 1-6 所示。

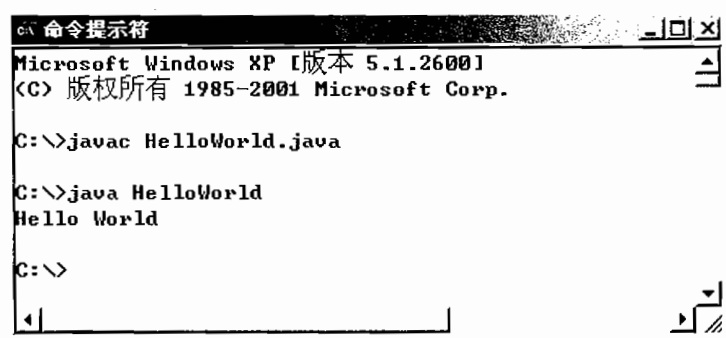


图 1-6 编译且执行 HelloWorld 程序

成功安装 J2SDK 1.4.2_03 之后，紧接下来安装 Tomcat 5.0.16。

1-2 安装 Tomcat 5.0.16

Tomcat 目前版本为 5.0.16，它是由 JavaSoft 和 Apache 开发团队共同提出合作计划(Apache Jakarta Project)下的产品。Tomcat 能支持 Servlet 2.4 和 JSP 2.0 并且是免费使用。

Tomcat 5.0.16 可以从 <http://jakarta.apache.org/tomcat/index.html> 网站自行免费下载，或者可以直接使用本书 CD 光盘中的 Tomcat 5.0.16，软件名称为：*jakarta-tomcat-5.0.16.exe*。

第一步：执行 *jakarta-tomcat-5.0.16.exe*（见图 1-7）；

先按【Next】，选择【I Agree】后，再按【Next】。

第二步：选择安装路径及安装内容（见图 1-8）；

通常我们会选择完全安装(Full)，即如图 1-8。在图 1-9【Tomcat】的选项中，主要有：Core、Service、Source Code 和 Documentation，假若选择安装 Service 时，尔后我们可以利用 Windows 的服务(控制面板 | 管理工具 | 服务)来设定重新开机启动时，Tomcat 能够自动启动。

JSP2.0 技术手册

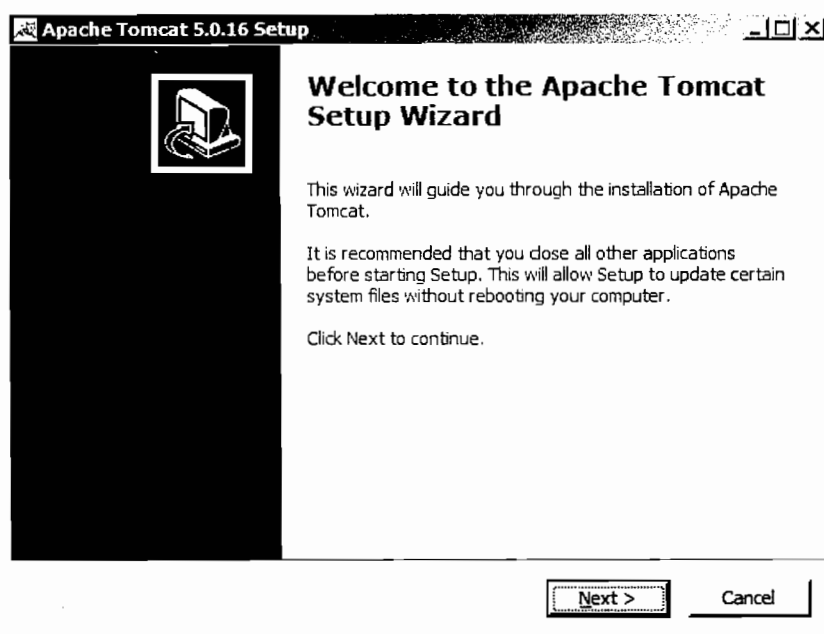


图 1-7 执行 jakarta-tomcat-5.0.16.exe

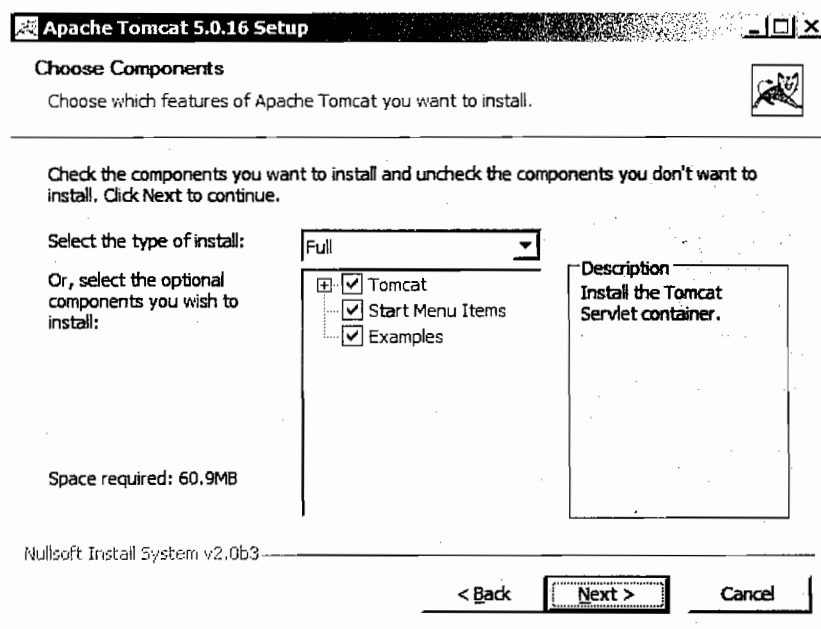


图 1-8 选择安装内容

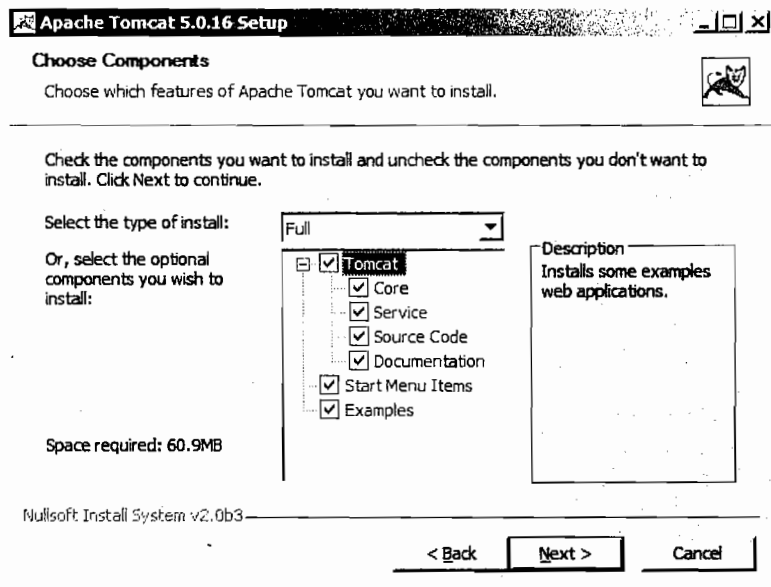


图 1-9 选择安装 Service

选择完全安装后，按【Next】。开始选择安装路径，如图 1-10 所示。

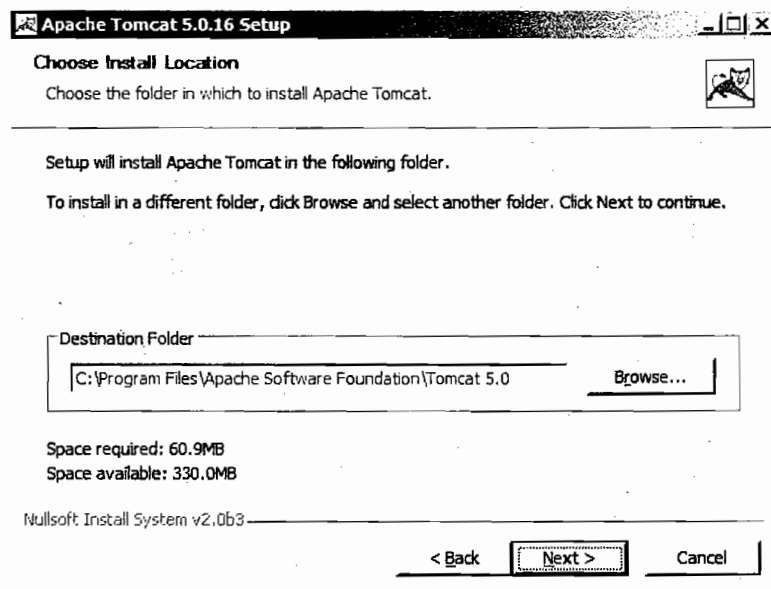


图 1-10 选择安装路径

第三步：设定 Tomcat Port 和 Administrator Login（见图 1-11）；

JSP2.0 技术手册

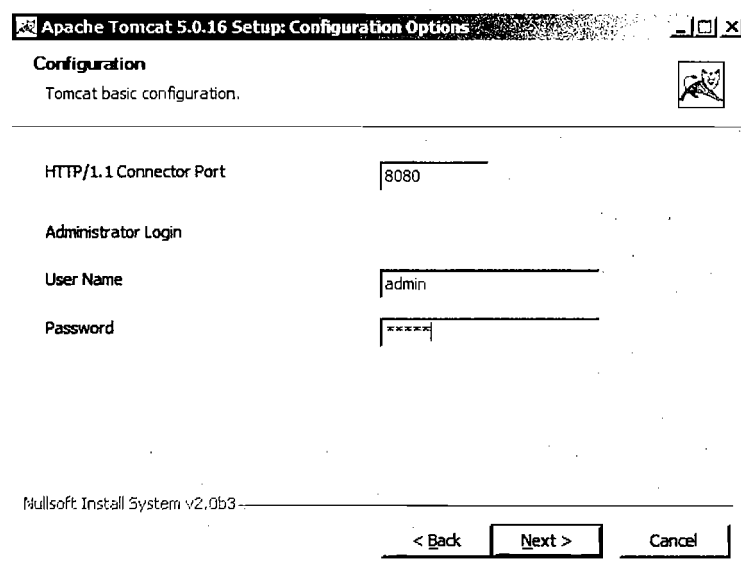


图 1-11 设定 Tomcat Port 和 Administrator Login

第四步：设定 Tomcat 使用的 JVM（见图 1-12）；

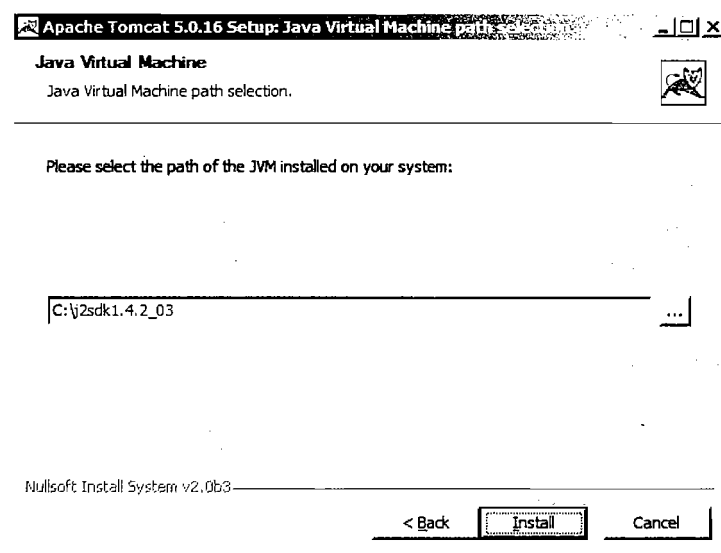


图 1-12 选择 Tomcat 使用的 JVM

确认无误后，按下【Install】，正式开始执行安装程序。安装成功后，会看到如图 1-13 的结果。

假若你勾选了【Run Apache Tomcat】，按下【Finish】之后，会直接启动 Tomcat 5.0.16，然后在你计算机的右下角，会出现绿色箭头的符号，如图 1-14。

JSP2.0 技术手册

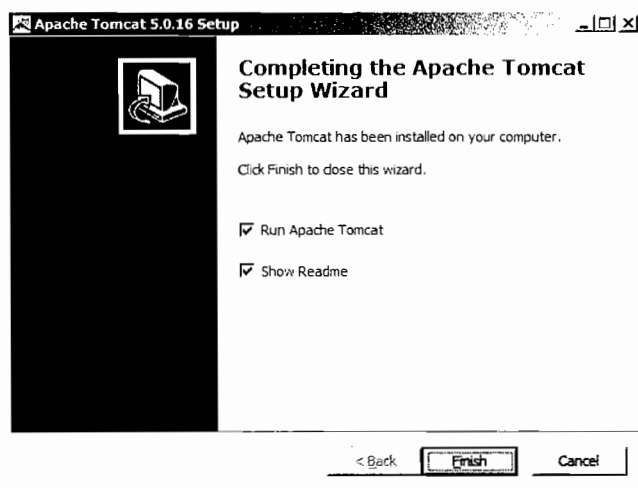


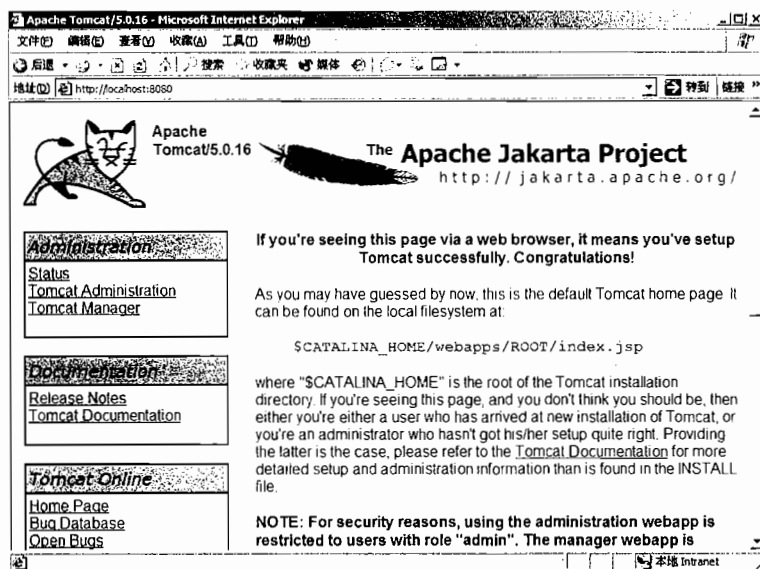
图 1-13 成功安装 Tomcat 5.0.16



图 1-14 Tomcat 图标

第五步：测试 Tomcat。

打开浏览器，如 IE，输入 <http://localhost:8080>，假若 Tomcat 安装成功，则会看到如图 1-15 的情形。

图 1-15 连接 <http://localhost:8080/>，测试 Tomcat 5.0.16

本书“第十二章：JSP 执行环境与开发工具”，对于 Tomcat 的使用及设定有更详细的介绍。

1-3 安装 JSPBook 站台范例

读者可以在 CD 光盘中找到本书的范例，程序文件名为 *JSPBook.war*。

第一步：安装 *JSPBook.war*：

安装的方法很简单，只要将 *JSPBook.war* 移至 *{Tomcat_Install}\webapps* 目录下(例如：*C:\Program Files\Apache Software Foundation\Tomcat 5.0\webapps\JSPBook.war*)，然后 *JSPBook.war* 会自动被 Tomcat 解压缩成 JSPBook 的目录，如图 1-16。

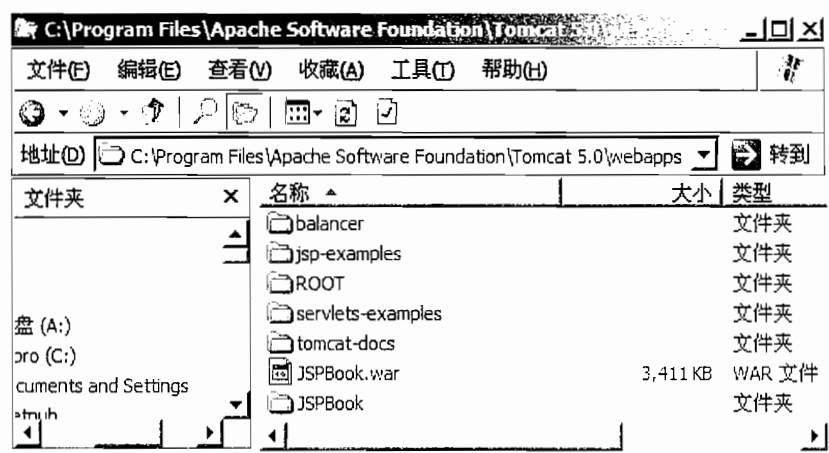


图 1-16 安装 JSPBook.war

第二步：设定 JSPBook 站台：

在 Tomcat 上建立一个 JSPBook 站台时，我们须修改 Tomcat 的 *server.xml* 文件，*server.xml* 位于 *{Tomcat_Install}\conf\server.xml* (例如：*C:\Program Files\Apache Software Foundation\Tomcat 5.0\conf\server.xml*)。

■ *server.xml*

```

.....
<!-- Tomcat Root Context -->
<!--
  <Context path="" docBase="ROOT" debug="0">
-->
  <Context path="/JSPBook" docBase="JSPBook" debug="0"
    crosscontext="true" reloadable="true" >
  </Context>
</Host>

```

JSP2.0 技术手册

```
</Engine>
</Service>
</Server>
```

这部分主要是设定 JSPBook 站台，其中 path="/JSPBook" 代表网域名称，即 http://IP_DomaninName/JSPBook；docBase="JSPBook" 代表站台的目录位置，即 {Tomcat_Install}\webapps\JSPBook；debug 则是设定 debug level，0 表示提供最少的信息，9 表示提供最多的信息；reloadable 则表示 Tomcat 在执行时，当 class、web.xml 被更新过时，都会自动重新加载，不需要重新启动 Tomcat。

注意

<Context>...</Context> 的位置必须在 <Host>...</Host> 之间，不可任意更动位置。

第三步：执行 JSPBook 站台（见图 1-17）：



图 1-17 执行 JSPBook 站台

第四步：JSPBook 站台目录结构。

JSPBook 目录下包含：

- (1) 各章节的 HTML/JSP 程序；
- (2) *dist* 目录：存放在 JSPBook 站台压缩后的 JSPBook.war；
- (3) *build.xml*：Ant 文件；
- (4) *WEB-INF* 目录：包含 *\classes*、*\lib*、*\tags* 和 *\src*；
- (5) *src* 目录：存放范例的源程序，如：JavaBean、Filter、Servlet，等等；
- (6) *Images* 目录：存放范例程序的图片。

图 1-18 为 JSPBook 站台目录结构。

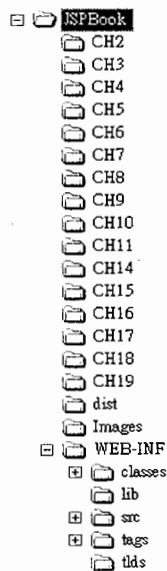


图 1-18 JSPBook 站台目录结构

1-4 安装 Ant 1.6

修改 JSP 程序时，Tomcat 会自动将 JSP 重新转译为 Servlet，并且编译 Servlet。但是，假若修改 Servlet、JavaBean 或 Filter 时，我们就必须自行先编译它们，然后再将它们重新部署至 *WEB-INF\classes* 下。

为了方便编译这些程序，笔者提供 JSPBook 站台的 *build.xml* 文件，因此，建议读者先安装 Ant 1.6，并且学习使用 Ant。

目前 Ant 的最新版本为 1.6，读者可以自行至 <http://ant.apache.org> 下载最新版本，如图 1-19 所示，或者直接使用本书 CD 光盘中的 Ant 1.6，软件名称为：*apache-ant-1.6.0-bin.zip*。

第一步：将 *apache-ant-1.6.0-bin.zip* 解压缩；

解压缩 *apache-ant-1.6.0-bin.zip* 之后，在 *apache-ant-1.6.0-bin* 目录下会有一目录 *apache-ant-1.6.0*，然后将 *apache-ant-1.6.0* 整个目录搬移至 C:\ 底下。

第二步：设定 Ant 1.6（见图 1-20）；

从【开始】→【设定】→【控制面板】→【系统】→【高级】→【环境变量】→【系统变量】，然后到【新建】。

```
ANT_HOME = C:\apache-ant-1.6.0
PATH = %ANT_HOME%\bin
```

第三步：测试 Ant 1.6；



图 1-19 <http://ant.apache.org>

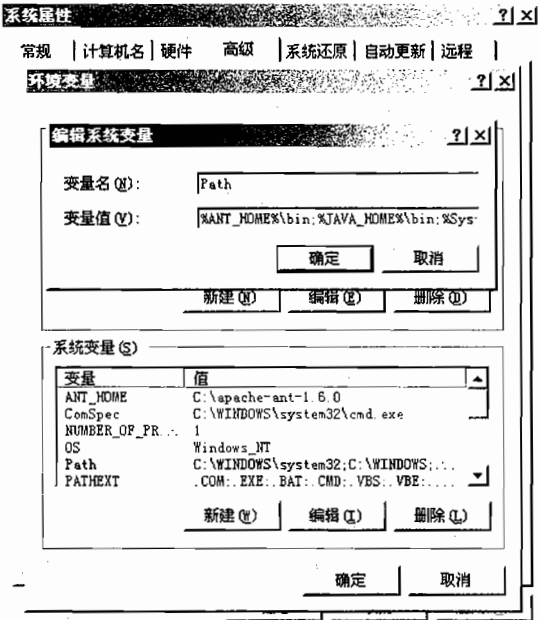


图 1-20 设定 Ant 1.6

打开命令提示符，输入 `ant -version`，假若执行成功，则会有如图 1-21 所示的结果。

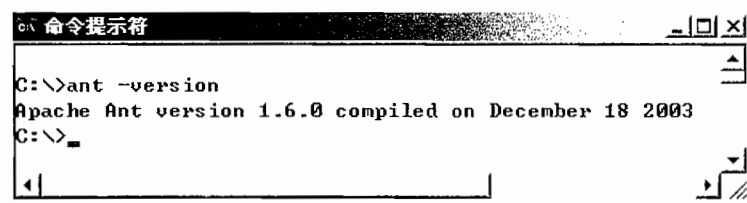


图 1-21 测试 Ant 1.6

第四步：使用 Ant 1.6 编译 `JSPBook\WEB-INF\src` 中的程序。

要编译修改过的 `JSPBook\WEB-INF\src` 中的程序时，首先打开命令提示符，移至 JSPBook 站台的所在目录，例如：`C:\Program Files\Apache Software Foundation\Tomcat 5.0\webapps\JSPBook`。

然后执行 `ant`，它会先自动找到 `JSPBook\build.xml` 文件，根据 `build.xml` 的设定，编译 `C:\Program Files\Apache Software Foundation\Tomcat 5.0\webapps\JSPBook\WEB-INF\src` 目录下所有的 Java 源文件，然后将产生的类文件存放至 `C:\Program Files\Apache Software Foundation\Tomcat 5.0\webapps\JSPBook\WEB-INF\classes` 目录下。图 1-22 为执行 `ant` 指令的结果。

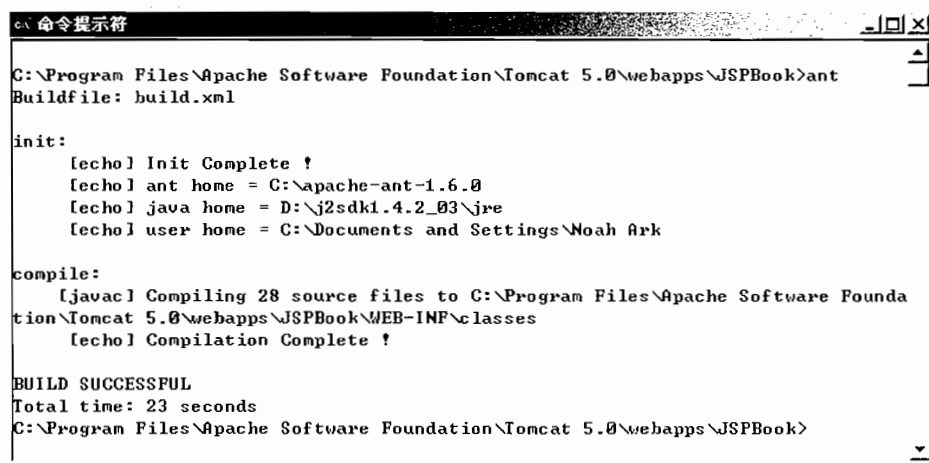


图 1-22 执行 ant 指令

2

第二章

Servlet 2.4 简介

在 JSP 技术尚未问世之前，Servlet 和 Applet 分别是 Java 在服务器端 (Server-Side) 和客户端 (Client-Side) 所推广的解决方案，两者相辅相成，各占有重要的地位。

JSP 的出现，弥补了 Servlet 在开发 Web-based 系统不足的地方。因此，笔者强烈建议读者，假若要对 JSP 有更深入的了解，Servlet 是最基本的内容，只要能够把 Servlet 学好，就更能够理解 JSP 技术底层运作的方式。

本章将分 5 节来为读者介绍 Servlet 2.4：

- 2-1 Servlet 简介
- 2-2 First Servlet Sample Code
- 2-3 Servlet 的生命周期
- 2-4 Servlet 范例程序
- 2-5 Servlet 2.4 的新功能

JSP2.0 技术手册

2-1 Servlet 简介

自 1997 年 3 月 Sun Microsystems 公司所组成的 JavaSoft 部门将 Servlet API 定案以来，推出了 Servlet API 1.0。就当时功能来说，Servlet 所提供的功能包含了当时的 CGI (Common Gateway Interface) 与 Netscape Server API (NSAPI) 之类产品的功能。

发展至今，Servlet API 的最新版本为 2.4 版。它依旧是一个具有跨平台特性、100% Pure Java 的 Server-Side 程序，它相对于在 Client 端执行的 Applet。Servlet 不只限于 HTTP 协议，开发人员可以利用 Servlet 自定义或延伸任何支持 Java 的 Server —— 包括 Web Server、Mail Server、Ftp Server、Application Server 或任何自定义的 Server。

Servlet 有以下优点：

● 可移植性(Portability)

Servlet 皆是利用 Java 语言来开发的，因此，延续 Java 在跨平台上的表现，不论 Server 的操作系统是 Windows、Solaris、Linux、HP-UX、FreeBSD、Compaq Tru 64、AIX 等等，都能够将我们所写好的 Servlet 程序放在这些操作系统上执行。借助 Servlet 的优势，就可以真正达到 Write Once, Serve Anywhere 的境界，这正是从事 Java 程序员最感到欣慰也是最骄傲的地方。

当程序员在开发 Applet 时，时常为了“可移植性”(portability)让程序员感到绑手绑脚的，例如：开发 Applet 时，为了配合 Client 端的平台（即浏览器版本的不同，plug-in 的 JDK 版本也不尽相同），达到满足真正“跨平台”的目的时，需要花费程序员大量时间来修改程序，为的就是能够让用户皆能够执行。但即使如此，往往也只能满足大部分用户，而其他少数用户，若要执行 Applet，仍须先安装合适的 JRE (Java Runtime Environment)。

但是 Servlet 就不同了，主要原因在于 Servlet 是在 Server 端上执行的，所以，程序员只要专心开发能在实际应用的平台环境下测试无误即可。除非你是从事做 Servlet Container 的公司，否则不须担心写出来的 Servlet 是否能在所有的 Java Server 平台上执行。

● 强大的功能

Servlet 能够完全发挥 Java API 的威力，包括网络和 URL 存取、多线程 (Multi-Thread)、影像处理、RMI (Remote Method Invocation)、分布式服务器组件 (Enterprise Java Bean)、对象序列化 (Object Serialization) 等。若想写个网络目录查询程序，则可利用 JNDI API，想连接数据库，则可利用 JDBC，有这些强大功能的 API 做后盾，相信 Servlet 更能够发挥其优势。

● 性能

Servlet 在加载执行之后，其对象实体(instance)通常会一直停留在 Server 的内存中，若有请求(request)发生时，服务器再调用 Servlet 来服务，假若收到相同服务的请求时，Servlet 会利用不同的线程来处理，不像 CGI 程序必须产生许多进程 (process)来处理数据。在性能的表现上，大大超越以往撰写的 CGI 程序。最后补充一点，那就是 Servlet 在执行时，不是一直停留在内存中，服务器会自动将停留时间过长一直没有执行的 Servlet 从内存中移除，不过有时候也可以自

JSP2.0 技术手册

行写程序来控制。至于停留时间的长短通常和选用的服务器有关。

● 安全性

Servlet 也有类型检查(Type Checking)的特性,并且利用 Java 的垃圾收集(Garbage Collection)与没有指针的设计,使得 Servlet 避免内存管理的问题。

由于在 Java 的异常处理(Exception-Handling)机制下,Servlet 能够安全地处理各种错误,不会因为发生程序上逻辑错误而导致整体服务器系统的毁灭。例如:某个 Servlet 发生除以零或其他不合法的运算时,它会抛出一个异常(Exception)让服务器处理,如:记录在记录文件中(log file)。

2-2 First Servlet Sample Code

■ HelloServlet.java

```
package tw.com.javaworld.CH2;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class HelloServlet extends HttpServlet {

    //Initialize global variables
    public void init(ServletConfig config) throws ServletException {
        super.init(config);
    }

    //Process the HTTP Get request
    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=GB2312");
        PrintWriter out = response.getWriter();

        out.println("<html>");
        out.println("<head><title>CH2 - HelloServlet</title></head>");
        out.println("<body>");
        out.println(" Hello World <br>");
        out.println("大家好");
        out.println("</body>");
        out.println("</html>");

        out.close();
    }

    //Get Servlet information
    public String getServletInfo() {
        return "tw.com.javaworld.CH2.HelloServlet Information";
    }
}
```

JSP2.0 技术手册

注意

HelloServlet.java 范例程序位于 JSPBook\WEB-INF\src\tw\com\javaworld\CH2，其中 JSPBook 范例程序的安装方法，请参见 1-3 节“安装 JSPBook 站台范例”和 1-4 节“安装 Ant 1.6”。

一开始我们必须导入(import) javax.servlet.*、javax.servlet.http.*。

javax.servlet.* : 存放与 HTTP 协议无关的一般性 Servlet 类；

javax.servlet.http.* : 除了继承 javax.servlet.* 之外，并且还增加与 HTTP 协议有关的功能。

所有 Servlet 都必须实现 javax.servlet.Servlet 接口(Interface)，但是通常我们都会从 javax.servlet.GenericServlet 或 javax.servlet.http.HttpServlet 择一来实现。若写的 Servlet 程序和 HTTP 协议无关，那么必须继承 GenericServlet 类，若有关，就必须继承 HttpServlet 类。

javax.servlet.* 里的 ServletRequest 和 ServletResponse 接口提供存取一般的请求和响应；而 javax.servlet.http.* 里的 HttpServletRequest 和 HttpServletResponse 接口，则提供 HTTP 请求及响应的存取服务。

```
public void init(ServletConfig config) throws ServletException {
    super.init(config);
}
```

这个例子中，一开始和 Applet 一样，也有 init() 的方法。当 Servlet 被 Container 加载后，接下来就会先执行 init() 的内容，因此，我们通常利用 init() 来执行一些初始化的工作。

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<html>");
    out.println("<head><title>CH2 - HelloServlet</title></head>");
    out.println("<body>");
    out.println("Hello World <br>");
    out.println("大家好");
    out.println("</body>");
    out.println("</html>");
    out.close();
}
```

Servlet 可以利用 HttpServletResponse 类的 setContentType() 方法来设定内容类型，我们要显示为 HTML 网页类型，因此，内容类型设为 "text/html"，这是 HTML 网页的标准 MIME 类型值。之后，Servlet 用 getWriter() 方法取得 PrintWriter 类型的 out 对象，它与 PrintStream 类似，但是它能对 Java 的 Unicode 字符进行编码转换。最后，再利用 out 对象把 "Hello World" 的字符串显示在网页上。

```
public void destroy() {
    .....
    ..... Servlet 结束时，会自动调用执行的程序 .....
    .....
}
```

JSP2.0 技术手册

若当 Container 结束 Servlet 时, 会自动调用 `destroy()`, 因此, 我们通常利用 `destroy()` 来关闭资源或是写入文件, 等等。

编译 `HelloServlet.java` 的方法:

(1) 将 `servlet-api.jar` 加入至 CLASSPATH 之中, 直接使用 `javac` 来编译 `HelloServlet.java`。其中 `servlet-api.jar` 可以在 `{Tomcat_Install}\common\lib` 找到。

(2) 直接使用 Ant 方式编译 `HelloServlet.java`, 请参见 1-4 节 “安装 Ant 1.6”。

编译好 `HelloServlet.java` 之后, 再来设定 `web.xml`, 如下:

```
<servlet>
  <servlet-name>HelloServlet</servlet-name>
  <servlet-class>tw.com.javaworld.CH2.HelloServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>HelloServlet</servlet-name>
  <url-pattern>/HelloServlet</url-pattern>
</servlet-mapping>
```

最后, `HelloServlet.java` 的执行结果如图 2-1 所示。



图 2-1 `HelloServlet.java` 的执行结果

顺利完成第一个 Servlet 程序后, 不知道读者有没有发现, 在 `HelloServlet.java` 主程序中, 其实大部分都是一些用来显示 HTML 的 `out.println("...")` 程序代码, 这就是 Servlet 用在开发 Web-based 系统时最麻烦的地方。假若 Servlet 要显示表格统计图时, 我想那时候程序员一定会疯掉, 因为你会发现, 其实你所有的时间都在 `out.println()`, 因此, Servlet 适合在简单的用户接口 (User Interface) 系统中。不过, 幸好有 JSP 技术来解决这项极为不方便的问题。

2-3 Servlet 的生命周期

当 Servlet 加载 Container 时, Container 可以在同一个 JVM 上执行所有 Servlet, 所以 Servlet 之间可以有效地共享数据, 但是 Servlet 本身的私有数据亦受 Java 语言机制保护。

JSP2.0 技术手册

Servlet 从产生到结束的流程如图 2-2 所示。

- (1) 产生 Servlet，加载到 Servlet Engine 中，然后调用 `init()` 这个方法来进行初始化工作。
- (2) 以多线程的方式处理来自 Client 的请求。
- (3) 调用 `destroy()` 来销毁 Servlet，进行垃圾收集 (garbage collection)。

Servlet 生命周期的定义，包括如何加载、实例化、初始化、处理客户端请求以及如何被移除。这个生命周期由 `javax.servlet.Servlet` 接口的 `init()`、`service()` 和 `destroy()` 方法表达。

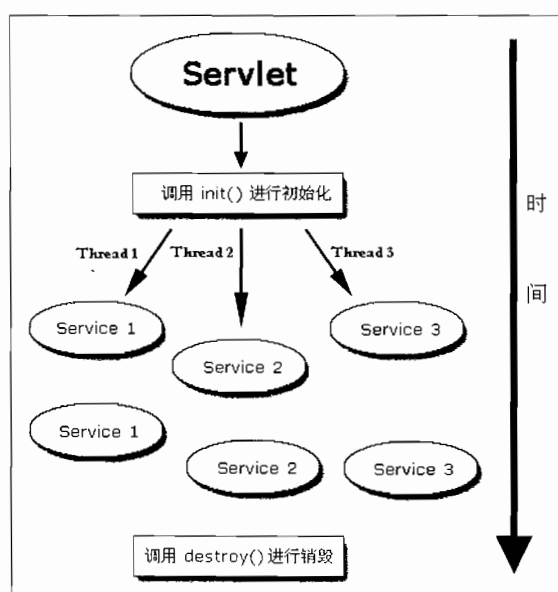


图 2-2 Servlet 从产生到结束的流程

1. 加载和实例化

当 Container 一开始启动,或是客户端发出请求服务时,Container 会负责加载和实例化一个 Servlet。

2. 初始化

Servlet 加载并实例化后,再来 Container 必须初始化 Servlet。初始化的过程主要是读取配置信息(例如 JDBC 连接)或其他须执行的任务。我们可以借助 `ServletConfig` 对象取得 Container 的配置信息,例如:

```
<servlet>
  <servlet-name>HelloServlet</servlet-name>
  <servlet-class>tw.com.javaworld.CH2.HelloServlet</servlet-class>
  <init-param>
    <param-name>user</param-name>
    <param-value>browser</param-value>
  </init-param>
</servlet>
```

JSP2.0 技术手册

其中 `user` 为初始化的参数名称; `browser` 为初始化的值。因此, 可以在 `HelloServlet` 程序中使用 `ServletConfig` 对象的 `getInitParameter("user")` 方法来取得 `browser`。

3. 处理请求

`Servlet` 被初始化后, 就可以开始处理请求。每一个请求由 `ServletRequest` 对象来接收请求; 而 `ServletResponse` 对象来响应该请求。

4. 服务结束

当 `Container` 没有限定一个加载的 `Servlet` 能保存多长时间, 因此, 一个 `Servlet` 实例可能只在 `Container` 中存活几毫秒, 或是其他更长的任意时间。一旦 `destroy()` 方法被调用时, `Container` 将移除该 `Servlet`, 那么它必须释放所有使用中的任何资源, 若 `Container` 需要再使用该 `Servlet` 时, 它必须重新建立新的实例。

2-4 Servlet 范例程序

为了说明 `Servlet` 和网页是如何沟通的, 笔者在此举一个 `Sayhi` 的范例程序。这个范例程序分为两部分: `Sayhi.html` 和 `Sayhi.java`。

在 `Sayhi.html` 中, 用户可以填入姓名, 然后按下【提交】后, 将数据传到 `Sayhi.java` 做处理, 而 `Sayhi.java` 负责将接收到的数据显示到网页上。

■ Sayhi.html

```
<html>
<head>
  <title>CH2 - Sayhi.html</title>
  <meta http-equiv="Content-Type" content="text/html; charset=GB2312">
</head>
<body>

<h2>Servlet 范例程序</h2>
<form name="Sayhi" Method="Post" action="/JSPBook/CH2/Sayhi" >

  <p>请访问者输入姓名: <input type="text" name="Name" size="30"></p>

  <input type="submit" value="提交">
  <input type="reset" value="清除">

</form>

</body>
</html>
```

`Sayhi.html` 的执行结果如图 2-3 所示。



图 2-3 Sayhi.html 的执行结果

■ Sayhi.java

```
package tw.com.javaworld.CH2;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class Sayhi extends HttpServlet {

    //Initialize global variables
    public void init(ServletConfig config) throws ServletException {
        super.init(config);
    }

    //Process the HTTP Get request
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=GB2312");
        PrintWriter out = response.getWriter();

        request.setCharacterEncoding("GB2312");
        String Name = request.getParameter("Name");

        out.println("<html>");
        out.println("<head><title>CH2 - Sayhi</title></head>");
        out.println("<body>");
        out.println("Hi: " + Name);
        out.println("</body>");
        out.println("</html>");
        out.close();
    }

    //Get Servlet information
    public String getServletInfo() {
        return "tw.com.javaworld.CH2.Sayhi Information";
    }
}
```

JSP2.0 技术手册

```
public void destroy() {  
    }  
}
```

从 *Sayhi.java* 的程序当中,可以发现 Servlet 是利用 `HttpServletRequest` 类的 `getParameter()` 方法来取得由网页传来的数据。不过数据通过 HTTP 协议传输时会被转码,因此在接收时,必须再做转码的工作,才能够正确地接收到数据。下面这段程序是做转码的动作:

```
request.setCharacterEncoding("GB2312");
```

编译 *Sayhi.java* 之后,再来设定 *web.xml*:

```
<servlet>  
    <servlet-name>Sayhi</servlet-name>  
    <servlet-class>tw.com.javaworld.CH2.Sayhi</servlet-class>  
</servlet>  
<servlet-mapping>  
    <servlet-name>Sayhi</servlet-name>  
    <url-pattern>/CH2/Sayhi</url-pattern>  
</servlet-mapping>
```

执行 `http://localhost:8080/JSPBook/CH2/Sayhi`, 结果如图 2-4 所示。



图 2-4 Sayhi.html 按下【提交】后,经过 Sayhi.java 处理后的结果

2-5 Servlet 2.4 的新功能

2003 年 11 月底, J2EE 1.4 规范正式发布, Servlet 也从原本的 2.3 版升级至 2.4 版。其中主要新增的功能有以下三点:

- (1) *web.xml* DTD 改用 XML Schema;
- (2) 新增 Filter 四种设定;
- (3) 新增 Request Listener、Event 和 Request Attribute Listener、Event。

2-5-1 web.xml 改用 XML Schema

Servlet 在 2.4 版之前, *web.xml* 都是使用 DTD(Document Type Definition)来定义 XML 文件

JSP2.0 技术手册

内容结构的，因此，Servlet 2.3 版 *web.xml* 一开始的声明如下：

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  .....
</web-app>
```

到了 Servlet 2.4 版之后，*web.xml* 改为使用 *XML Schema*，此时 *web.xml* 的声明如下：

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
  version="2.4">
  .....
</web-app>
```

由 DTD 改为 Schema，主要加强两项功能：

- (1) 元素可不依照顺序设定；
- (2) 更强大的验证机制。

下面的范例，在 Servlet 2.3 版是不合规则的 *web.xml* 文件：

```
<web-app>
  ...
  <servlet>
    <servlet-name>ServletA</servlet-name>
    <servlet-class>tw.com.javaworld.servlet.ServletA</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>ServletA</servlet-name>
    <url-pattern>/ServletA/*</url-pattern>
  </servlet-mapping>

  <servlet>
    <servlet-name>ServletB</servlet-name>
    <servlet-class> tw.com.javaworld.servlet.ServletB</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>ServletB</servlet-name>
    <url-pattern>/ServletB /*</url-pattern>
  </servlet-mapping>
  ...
</web-app>
```

因为<servlet-mapping>元素必须在<servlet>元素之后，因此，上述的范例要改为：

```

<web-app>
...
<servlet>
  <servlet-name>ServletA</servlet-name>
  <servlet-class>tw.com.javaworld.servlet.ServletA</servlet-class>
</servlet>

<servlet>
  <servlet-name>ServletB</servlet-name>
  <servlet-class> tw.com.javaworld.servlet.ServletB</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>ServletA</servlet-name>
  <url-pattern>/ServletA/*</url-pattern>
</servlet-mapping>

<servlet-mapping>
  <servlet-name>ServletB</servlet-name>
  <url-pattern>/ServletB /*</url-pattern>
</servlet-mapping>
...
</web-app>

```

不过在 Servlet 2.4 版之后，原来的范例也算是一个合法的 *web.xml* 文件，不再须注意元素的顺序。

除此之外，Servlet 2.4 版 *web.xml* 的 Schema 更能提供强大的验证机制，例如：

(1) 可检查元素的值是否为合法的值。例如：<filter-mapping>的<dispatcher>元素，其值只能为 REQUEST、FORWARD、INCLUDE 和 ERROR，如下所示：

```

<filter-mapping>
  <filter-name>Hello</filter-name>
  <url-pattern>/CH11/*</url-pattern>
  <dispatcher>REQUEST</dispatcher>
  <dispatcher>FORWARD</dispatcher>
</filter-mapping>

```

若<dispatcher>元素的值不为上述四种时，此 *web.xml* 将会发生错误。

(2) 可检查如 Servlet、Filter 或 EJB-ref 等等元素的名称是否惟一。例如：

```

<servlet>
  <servlet-name>ServletA</servlet-name>
  <servlet-class>tw.com.javaworld.servlet.ServletA</servlet-class>
</servlet>

<servlet>
  <servlet-name>ServletA</servlet-name>
  <servlet-class>tw.com.javaworld.servlet.ServletB</servlet-class>
</servlet>

```

(3) 可检查元素值是否为合法文字字符或数字字符。例如：

JSP2.0 技术手册

```
<filter-mapping>
    <filter-name>Hello</filter-name>
    <url-pattern>/CH11/*</url-pattern>
</filter-mapping>
```

2-5-2 新增 Filter 四种设定

Servlet 2.3 版新增了 Filter 的功能，不过它只能由客户端发出请求来调用 Filter，但若使用 RequestDispatcher.forward()或 RequestDispatcher.include()的方法调用 Filter 时，Filter 却不会执行。因此，在 Servlet 2.4 版中，新增 Filter 的设定<dispatcher>来解决这个问题。有关 Filter 的部分在本书“第十一章：Filter 与 Listener”有更详细的介绍。

Servlet 2.4 版新增的 Filter 四种设定为：REQUEST、FORWARD、INCLUDE 和 ERROR。假若你有一个 SimpleFilter，它只允许由客户端发出请求或由 RequestDispatcher.include()的方式来调用执行 SimpleFilter，此时 SimpleFilter 的设定如下：

```
<filter>
    <filter-name>SimpleFilter</filter-name>
    <filter-class>tw.com.javaworld.CH11.SimpleFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>SimpleFilter</filter-name>
    <url-pattern>/CH11/*</url-pattern>
    <dispatcher>REQUEST</dispatcher>
    <dispatcher>INCLUDE</dispatcher>
</filter-mapping>
```

2-5-3 新增 Request Listener、Event 和 Request Attribute Listener、Event

在 Servlet 2.3 版中，新增许多的 Listener 接口和 Event 类（见表 2-1）：

表 2-1

Listener 接口	Event 类
ServletContextListener	ServletContextEvent
ServletContextAttributeListener	ServletContextAttributeEvent
HttpSessionListener	HttpSessionEvent
HttpSessionActivationListener	
HttpSessionAttributeListener	

在 Servlet 2.4 版陆续又多新增 Request Listener、Event 和 Request Attribute Listener、Event（见表 2-2）：

表 2-2

Listener 接口	Event 类
ServletRequestListener	ServletRequestEvent
ServletRequestAttributeListener	ServletRequestAttributeEvent

这部分在“第十一章：Filter 与 Listener”中有更详细的介绍。

2-5-4 Servlet 2.4 的其他变更

Servlet 2.4 其他较显著的变更如：

(1) 取消 SingleThreadModel 接口。当 Servlet 实现 SingleThreadModel 接口时，它能确保同时间内，只能有一个 thread 执行此 Servlet。

(2) <welcome-file-list>可以为 Servlet。例如：

```
<servlet>
  <servlet-name>Index</servlet-name>
  <servlet-class>tw.com.javaworld.IndexServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>Index</servlet-name>
  <url-pattern>/*</url-pattern>
</servlet-mapping>
<welcome-file-list>
  <welcome-file>Index</welcome-file>
</welcome-file-list>
```

(3) ServletRequest 接口新增一些方法，如：

```
public String getLocalName( );
public String getLocalAddr( );
public int getLocalPort( );
public int getRemotePort( );
```


3

第三章

JSP 2.0 简介

本章开始要进入我们的主题——JavaServer Pages。主要目的是建立对 JSP 技术的初步认识，本章将分为 6 节来为读者介绍：

- 3-1 JavaServer Pages 技术
- 3-2 What is JSP
- 3-3 JSP 与 Servlet 的比较
- 3-4 JSP 的执行过程
- 3-5 JSP 与 ASP 和 ASP+的比较
- 3-6 JSP 2.0 新功能

JSP2.0 技术手册

3-1 JavaServer Pages 技术

JavaServer Pages 技术是一个纯 Java 平台的技术，它主要用来产生动态网页内容，包括：HTML、DHTML、XHTML 和 XML。JavaServer Pages 技术能够让网页人员轻易建立起功能强大、有弹性的动态内容。

JavaServer Pages 技术有下列优点：

● Write Once, Run Anywhere 特性

作为 Java 平台的一部分，JavaServer Pages 技术拥有 Java 语言“一次编写，各处执行”的特点。随着越来越多的供货商将 JavaServer Pages 技术添加到他们的产品中，您可以针对自己公司的需求，做出审慎评估后，选择符合公司成本及规模的服务器，假若未来的需求有所变更时，更换服务器平台并不影响之前所投下的成本、人力所开发的应用程序。

● 搭配可重复使用的组件

JavaServer Pages 技术可依赖于重复使用跨平台的组件（如：JavaBean 或 Enterprise JavaBean 组件）来执行更复杂的运算、数据处理。开发人员能够共享开发完成的组件，或者能够加强这些组件的功能，让更多用户或是客户团体使用。基于善加利用组件的方法，可以加快整体开发过程，也大大降低公司的开发成本和人力。

● 采用标签化页面开发

Web 网页开发人员不一定是熟悉 Java 语言的程序员。因此，JSP 技术能够将许多功能封装起来，成为一个自定义的标签，这些功能是完全根据 XML 的标准来制订的，即 JSP 技术中的标签库(Tag Library)。因此，Web 页面开发人员可以运用自定义好的标签来达成工作需求，而无须再写复杂的 Java 语法，让 Web 页面开发人员亦能快速开发出一动态内容网页。

今后，第三方开发人员和其他人员可以为常用功能建立自己的标签库，让 Web 网页开发人员能够使用熟悉的开发工具，如同 HTML 一样的标签语法来执行特定功能的工作。本书将在“第十五章：JSP Tag Library”和“第十六章：Simple Tag 与 Tag File”中详细地为各位介绍如何制作标签。

● N-tier 企业应用架构的支持

有鉴于网际网络的发展，为因应未来服务越来越繁杂的要求，且不再受地域的限制，因此，必须放弃以往 Client-Server 的 Two-tier 架构，进而转向更具威力、弹性的分散性对象系统。由于 JavaServer Page 技术是 Java 2 Platform Enterprise Edition (J2EE) (相关信息请参阅 www.javasoft.com/products/j2ee)集成中的一部分，它主要是负责前端显示经过复杂运算后之结果内容，而分散性的对象系统则是主要依赖 EJB (Enterprise JavaBean)和 JNDI (Java Naming and Directory Interface)构建而成。

3-2 What is JSP

JSP(JavaServer Pages)是由 Sun 公司倡导、许多别的公司参与一起建立的一种新动态网页技术标准,类似其他技术标准,如 ASP、PHP 或是 ColdFusion,等等。

在传统的网页 HTML 文件(*.htm,*.html)中加入 Java 程序片段(Scriptlet)和 JSP 标签,构成了 JSP 网页(*.jsp)。Servlet/JSP Container 收到客户端发出的请求时,首先执行其中的程序片段,然后将执行结果以 HTML 格式响应给客户端。其中程序片段可以是:操作数据库、重新定向网页以及发送 E-Mail 等等,这些都是建立动态网站所需要的功能。所有程序操作都在服务器端执行,网络上传送给客户端的仅是得到的结果,与客户端的浏览器无关,因此,JSP 称为 Server-Side Language。

3-3 JSP 与 Servlet 的比较

Sun 公司首先发展出 Servlet,其功能非常强大,且体系设计也很完善,但是它输出 HTML 语法时,必须使用 `out.println()`一句一句地输出,例如下面一段简单的程序:

```
out.println("<html>");
out.println("<head><title>demo1</title></head>");
out.println(" Hello World <br>");
out.println("<body>");
out.println("大家好");
out.println("</body>");
out.println("</html>");
```

由于这是一段简单的 Hello World 程序,还看不出来其复杂性,但是当整个网页内容非常复杂时,那么你的 Servlet 程序可能大部分都是用 `out.println()`输出 HTML 的标签了!

后来 Sun 公司推出类似于 ASP 的嵌入型 Scripting Language,并且给它一个新的名称:JavaServer Pages,简称为 JSP。于是上面那段程序改为:

```
<html>
<head><title>www.javaworld.com.tw - 台湾 Java 论坛</title></head>
<body>
<%
    out.println(" Hello World <br>");
    out.println("大家好");
%>
</body>
</html>
```

这样就简化了 Web 网页程序员的负担,不用为了网页内容编排的更动,又需要由程序员来做修改。

3-4 JSP 的执行过程

在介绍 JSP 语法之前，先向读者说明一下 JSP 的执行过程（见图 3-1）。

- (1) 客户端发出 Request (请求);
- (2) JSP Container 将 JSP 转译成 Servlet 的源代码;
- (3) 将产生的 Servlet 的源代码经过编译后，并加载到内存执行;
- (4) 把结果 Response (响应)至客户端。

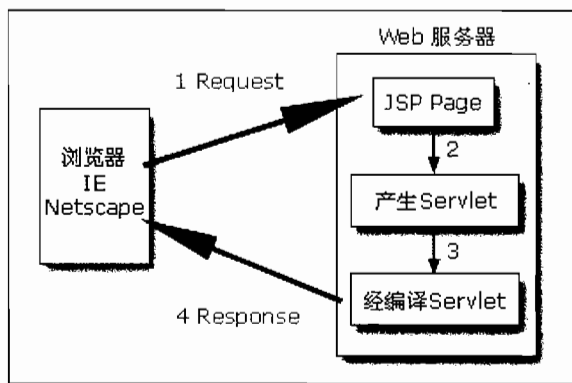


图 3-1 JSP 的执行过程

一般人都会以为 JSP 的执行性能会和 Servlet 相差很多，其实执行性能上的差别只在**第一次的执行**。因为 JSP 在执行第一次后，会被编译成 Servlet 的类文件，即为 `XXX.class`，当再重复调用执行时，就直接执行第一次所产生的 Servlet，而不用再重新把 JSP 编译成 Servlet。因此，除了第一次的编译会花较久的时间之外，之后 JSP 和 Servlet 的执行速度就几乎相同了。

在执行 JSP 网页时，通常可分为两个时期：转译时期(Translation Time)和请求时期(Request Time)（见图 3-2）。

转译时期：JSP 网页转译成 Servlet 类。

请求时期：Servlet 类执行后，响应结果至客户端。

补充

转译期间主要做了两件事情：将 JSP 网页转译为 Servlet 源代码(.java)，此段称为转译时期(Translation time); 将 Servlet 源代码(.java)编译成 Servlet 类(.class)，此段称为编译时期(Compilation time)。

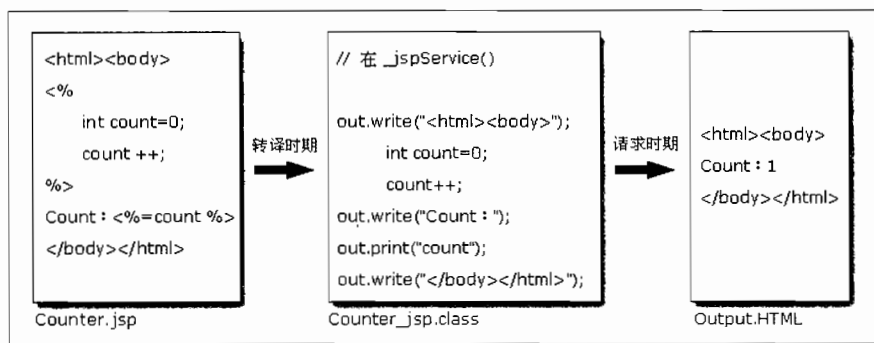


图 3-2 转译时期与请求时期程序图

当 JSP 网页在执行时, JSP Container 会做检查的工作, 若发现 JSP 网页有更新修改时, JSP Container 才会再次编译 JSP 成 Servlet; JSP 没有更新时, 就直接执行前面所产生的 Servlet。

笔者在这里以 Tomcat 为例, 看看 Tomcat 如何将 JSP 转译成 Servlet。首先笔者写一个简单的 JSP 网页 —— *HelloJSP.jsp*:

■ *HelloJSP.jsp*

```

<%@ page contentType="text/html;charset=GB2312" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<html>
<head>
  <title>CH3 - HelloJSP.jsp</title>
</head>
<body>

<h2>JSP 将会被转译为 Servlet</h2>

<%!
    int k = 0;
%>
<c:out value="Hi" />
<%
    String name = "browser";

    out.println("大家好 !!");
%>
<%= name %>

</body>
</html>
  
```

当执行 *HelloJSP.jsp* 时, Tomcat 会将它先转译为 Servlet。这个 Servlet 程序是放在 {Tomcat_Install}\Apache Software Foundation\Tomcat 5.0\work\Catalina\localhost\JSPBook\

org\apache\jsp\CH3 目录下的 *HelloJSP_jsp.java* 和 *HelloJSP_jsp.class*。其中 *HelloJSP_jsp.java* 就是 *HelloJSP.jsp* 所转译的 Servlet 源代码，它的程序如下：

■ *HelloJSP_jsp.java*

```
package org.apache.jsp.CH3;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;

public final class HelloJSP_jsp extends org.apache.jasper.runtime.HttpJspBase
    implements org.apache.jasper.runtime.JspSourceDependent {

    int k = 0;

    private static java.util.Vector _jspx_dependants;

    private org.apache.jasper.runtime.TagHandlerPool _jspx_tagPool_c_out_value;

    public java.util.List getDependants() {
        return _jspx_dependants;
    }

    public void _jspInit() {
        _jspx_tagPool_c_out_value =
            org.apache.jasper.runtime.TagHandlerPool.getTagHandlerPool(
                getServletConfig());
    }

    public void _jspDestroy() {
        _jspx_tagPool_c_out_value.release();
    }

    public void _jspService(HttpServletRequest request, HttpServletResponse
        response) throws java.io.IOException, ServletException {

        JspFactory _jspxFactory = null;
        PageContext pageContext = null;
        HttpSession session = null;
        ServletContext application = null;
        ServletConfig config = null;
        JspWriter out = null;
        Object page = this;
        JspWriter _jspx_out = null;

        try {
            _jspxFactory = JspFactory.getDefaultFactory();
            response.setContentType("text/html;charset=GB2312");
            pageContext = _jspxFactory.getPageContext(this, request, response,
                null, true, 8192, true);
```

JSP2.0 技术手册

```

    application = pageContext.getServletContext();
    config = pageContext.getServletConfig();
    session = pageContext.getSession();
    out = pageContext.getOut();
    _jspx_out = out;

    out.write("\r\n");
    out.write("\r\n\r\n");
    out.write("<html>\r\n");
    out.write("<head>\r\n ");
    out.write("<title>CH3 - HelloJSP.jsp");
    out.write("</title>\r\n");
    out.write("</head>\r\n");
    out.write("<body>\r\n\r\n");
    out.write("<h2>JSP 将会被转译为 Servlet");
    out.write("</h2>\r\n\r\n");
    out.write("\r\n");
    if (_jspx_meth_c_out_0(pageContext))
        return;
    out.write("\r\n");

    String name = "browser";

    out.println("大家好 !!");

    out.write("\r\n");
    out.print( name );
    out.write("\r\n\r\n");
    out.write("</body>\r\n");
    out.write("</html>");
} catch (Throwable t) {
    if (!(t instanceof SkipPageException)){
        out = _jspx_out;
        if (out != null && out.getBufferSize() != 0)
            out.clearBuffer();
        if (pageContext != null) pageContext.handlePageException(t);
    }
} finally {
    if (_jspxFactory != null) _jspxFactory.releasePageContext(pageContext);
}
}

private boolean _jspx_meth_c_out_0(PageContext pageContext)
    throws Throwable {
    JspWriter out = pageContext.getOut();
    // c:out
    org.apache.taglibs.standard.tag.rt.core.OutTag _jspx_th_c_out_0 =
        (org.apache.taglibs.standard.tag.rt.core.OutTag) _jspx_tagPool_c_out_value.
        get( org.apache.taglibs.standard.tag.rt.core.OutTag.class);
    _jspx_th_c_out_0.setPageContext(pageContext);
    _jspx_th_c_out_0.setParent(null);
    _jspx_th_c_out_0.setValue(new String("Hi"));
    int _jspx_eval_c_out_0 = _jspx_th_c_out_0.doStartTag();
    if (_jspx_th_c_out_0.doEndTag() ==

```

```

        javax.servlet.jsp.tagext.Tag.SKIP_PAGE)
        return true;
    _jspx_tagPool_c_out_value.reuse(_jspx_th_c_out_0);
    return false;
}
}

```

当 JSP 被转译成 Servlet 时，内容主要包含三部分：

```

public void _jspInit() {
    ... 略
}

public void _jspDestroy() {
    ... 略
}

public void _jspService(HttpServletRequest request, HttpServletResponse
    response) throws java.io.IOException, ServletException {
    ... 略
}

```

`_jspInit()`：当 JSP 网页一开始执行时，最先执行此方法。因此，我们通常会把初始化的工作写在此方法中。

`_jspDestroy()`：JSP 网页最后执行的方法。

`_jspService()`：JSP 网页最主要的程序都是在此方法中。

接下来笔者将 *HelloJSP.jsp* 和 *HelloJSP_jsp.java* 做一个简单的对照：

```
<%@ page contentType="text/html; charset=GB2312" %>
```



```
response.setContentType("text/html; charset=GB2312");
```

```
<%! int k = 0; %>
```



```
int k = 0; // 此为全局变量
```

```

<html>
<head>
    <title>CH3 - HelloJSP.jsp</title>
</head>
<body>

```

```
<h2>JSP 将会被转译为 Servlet</h2>
```



JSP2.0 技术手册


```

out.write("\r\n");
out.write("\r\n\r\n");
out.write("<html>\r\n");
out.write("<head>\r\n ");
out.write("<title>CH3 - HelloJSP.jsp");
out.write("</title>\r\n");
out.write("</head>\r\n");
out.write("<body>\r\n\r\n");
out.write("<h2>JSP 将会被转译为 Servlet");
out.write("</h2>\r\n\r\n");
out.write("\r\n");

```

```

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<c:out value="Hi" />

```



```

if (_jspx_meth_c_out_0(pageContext))
    return;
... 略
private boolean _jspx_meth_c_out_0(PageContext pageContext)
    throws Throwable {
    JspWriter out = pageContext.getOut();
    // c:out
    org.apache.taglibs.standard.tag.rt.core.OutTag _jspx_th_c_out_0 =
        (org.apache.taglibs.standard.tag.rt.core.OutTag) _jspx_tagPool_c_out_value.
        get(org.apache.taglibs.standard.tag.rt.core.OutTag.class);
    _jspx_th_c_out_0.setPageContext(pageContext);
    _jspx_th_c_out_0.setParent(null);
    _jspx_th_c_out_0.setValue(new String("Hi"));
    int _jspx_eval_c_out_0 = _jspx_th_c_out_0.doStartTag();
    if (_jspx_th_c_out_0.doEndTag() ==
        javax.servlet.jsp.tagext.Tag.SKIP_PAGE)
        return true;
    _jspx_tagPool_c_out_value.reuse(_jspx_th_c_out_0);
    return false;
}

```

```

<%
    String name = "browser";
    out.println("大家好 !!");
%>
<%= name %>

```



```

String name = "browser";
out.println("大家好 !!");
out.write("\r\n");
out.print( name );

```


3-5 JSP 与 ASP 和 ASP+ 的比较

■ JSP 与 ASP 的比较

一般说来, Sun 公司的 JavaServer Pages (JSP) 和 Microsoft 的 Active Server Pages (ASP) 在技术方面有许多相似之处。两者都为动态网页的技术, 并且双方都能够替代 CGI 技术, 使网站的开发时程能够大大缩短, 在性能上也有较高的表现, 更重要的一点是, 两者都能够为程序员提供组件设计的功能, 通过组件设计, 将网页中逻辑处理部分交由组件负责处理(ASP 使用 COM 组件、JSP 则有 JavaBean 组件), 而和网页上的排版、美工分离。

尽管 JavaServer Pages 技术和 Active Server Pages (ASP) 在许多方面都很相似, 但仍然存在很多不同之处, 其中本质上的区别在于: 两者是来源于不同的技术规范组织。以下就来比较两大技术有哪些不同点, 而又为各自带来哪些优势。

■ 平台和服务器的弹性

ASP (Active Server Pages) 技术主要在微软(Microsoft)公司的 Windows 平台上运行, 其中包括 Windows 2000、Windows XP 和 Windows 2003, 并且搭配其 WEB 服务器 IIS (Internet Information Services)。但是, 在其他的平台运行时, 不是性能低落, 就是根本不支持, 因此, 当在开发网站系统时, 选择 NT+IIS+ASP 的体系结构时, 未来当系统无法负荷时, 也只能继续选择 Windows 平台的服务器, 无法改写在性能表现相当优异的 UNIX 平台上。

JSP (JavaServer Pages) 技术主要运行在操作系统上的一个 Java Virtual Machine (JVM) 虚拟机上, 因此, 它能够跨越所有的平台, 例如: NT、Windows 2000、Solaris、Linux、OS/390、AIX、HP-UX, 等等, 除了能在各式各样的操作系统上执行, 并且能搭配现有的 WEB 服务器: Apache、IIS、Netscape Enterprise Server, 等等, 将静态的 HTML 网页交由执行速度较快的 Web Server 处理, 而动态产生网页的部分, 就交由 JSP Container 来执行。由上述可知, JSP (JavaServer Pages) 技术在跨平台的表现比 ASP 来得更有弹性。

WEB 网页程序员未来在开发电子商务平台时, 就不需要再考虑客户厂商的操作系统平台, 可更专心于系统功能的开发。相应地, 厂商在使用 JavaServer Pages 技术开发的系统平台时, 不再需要担心未来在扩充软、硬件时, 是否产生不兼容的问题。光这一点, 就能为企业省下一大笔的费用, 这是 JSP 的主要优点。

■ 语法结构

ASP 语法结构上, 是以 "<% " 和 "%>" 作为标记符号, 而 JSP 也是使用相同标记符号作为程序

JSP2.0 技术手册

的区段范围的。但不同的是，标记符号之间所使用的语言：ASP 为 JavaScript 或 VBScript；而 JSP 为 Java。Java 是有严格规划、强大且易扩充的语言，远优于 VBScript 语言。

Java 使程序员的工作在其他方面也变得一样容易、简单。例如：当 ASP 应用程序在 Windows NT 系统可能会造成系统 Crash (当机)时，由于 JSP 是在 JVM 上执行程序，且提供强大的异常事件处理机制，因此，不会因为程序撰写的疏忽，而导致服务器操作系统的损毁。

并且 Java 语言提供防止直接存取内存的功能，存取内存产生的错误，通常也正是造成服务器损毁的最主要原因之一。最后，最重要的原因，Java 语言是一个有严谨规范、有系统组织的语言，对一个专业的 Java 程序员来说，也真正达到 Learn Once, Write Anywhere(学一次，皆可开发)的境界。

■ 开放的开发环境

自从 1995 年，Sun 公司已经开放技术与国际 Java 组织合作开发和修改 Java 技术与规范。针对 JSP 的新技术，Sun 公司授权工具供货商（如 Macromedia）、同盟公司（如 Apache、Netscape）、协力厂商及其他公司。最近，Sun 公司将最新版本的 Servlet 2.4 和 JSP 2.0 的源代码发放给 Apache，以求 JSP 与 Apache 紧密地相互发展。Apache、Sun 和许多其他的公司及个人公开成立一个咨询机构，以便任何公司和个人都能免费取得信息。（详见：<http://jakarta.apache.org>）

JSP 应用程序接口 (API) 毫无疑问已经取得成功，并随着 Java 组织不断扩大其应用的范围，目前全力发展 Java 技术的厂商不胜枚举，例如：最近 IBM 公司强力推广的 WebSphere 家族，正是完全支持 J2EE 标准而开发。数据库厂商 Oracle 也发展自己的 Application Server 来和自己公司本身数据库产品 Oracle 9i 做一紧密的结合。那也更不用提 Amazon 系统的供货商 BEA 公司，它的产品 WebLogic 也是完全支持 JavaServer Pages 技术和 J2EE 规范的。

相反，ASP 技术仅依靠微软本身的推动，其发展建立在独占、封闭的基础之上，并且微软本身的技术又只允许在微软相关平台的服务器上执行，因此，在标准方面显得有点力不从心。

■ 语法的延展性

ASP 和 JSP 都使用标签与 Scripting Language 来制作动态 WEB 网页，JavaServer Pages 2.0 新规范中，能够让程序员自由扩展 JSP 标签来应用。JSP 开发者能自定义标签库 (Tag Library)，所以网页制作者能充分利用与 XML 兼容的标签技术强大的功能，大大减低对 Java 语法的依赖，并且也可以利用 XML 强大的功能，做到数据、文件格式的标准化。相关标签库请参考“第十五章：JSP Tag Library”，其中有更加完整的说明。

■ 执行性能表现

ASP 和 JSP 在执行性能的表现上,有一段显著的差距,JSP 除了在一开始加载的时间会比较久外,之后的表现就远远比 ASP 的表现来得好。原因在于:JSP 在一开始接受到请求时,会产生一份 Servlet 实体(instance),它会先被暂存在内存中,我们称之为持续(Persistence),当再有相同请求时,这实体会产生一个线程(thread)来服务它。如果过了一段时间都不再用到此实体时,Container 会自动将其释放,至于时间的长短,通常都是可以在 Container 上自行设定的。

而 ASP 在每次接收到请求时,都必须要重新编译,因此,JSP 的执行比每次都要编译执行的 ASP 要快,尤其是程序中存在循环操作时,JSP 的速度要快上 1 到 2 倍。不过,ASP 在这部分的缺陷,将随 ASP+ 的出现有所改观,在新版的 ASP+ 技术中,性能表现上有很大的突破。

■ JSP 与 ASP+ 的比较

1. 面向对象性

C# 为一种面向对象语言,从很多方面来看,C# 将给 ASP+ 带来类似于 Java 的功能,并且它和 Windows 环境紧密结合,因此,具备更快的性能。笔者认为,C# 是微软在市场上击败 Java 的主要工具。

2. 数据库连接

ASP 最大的优点是它使用 ADO 对象,因此,ASP Web 数据库应用开发特别简单。ASP+ 发展了更多的功能,因为有了 ADO+,ADO+ 带来了更强大更快速的功能。JSP 和 JDBC 目前在易用性和性能上和 ASP/ADO 相比已有些落后,当新版本 ASP+/ADO+ 出现后这样的差别会更明显,这部分希望 Sun 尽快追赶 ASP+/ADO+ 的组合。

3. 大型网站应用

ASP+ 将对大型网站有更好的支持。事实上,微软在这方面付出了巨大的努力,ASP+ 可以让你考虑到多服务器(multiple servers)的场合,当你需要更强大的功能时,仅仅只需要增加一台服务器。ASP+ 现在可以在大型项目方面与 JSP 一样具有等同的能力,而且 ASP+ 还有价格方面的优势,因为所有的组件将是服务器操作系统的一部分。

结论:

除了上述 ASP、ASP+ 和 JSP 之外,笔者再提供一篇在网络上 ASP 和 JSP 比较的文章:
<http://www.indiawebdevelopers.com/technology/Java/jsp.asp>,希望能带给读者更客观的评论。

除了 ASP 之外,PHP 和 ColdFusion 皆为近年来常用来开发 WEB 动态网页内容的工具,各开发工具皆有其优、缺点。ASP 和 PHP 最大的好处就是开发中、小型网站非常快速,市面上的

JSP2.0 技术手册

书籍也较多,学习起来能较快上手。尤其因为 PHP 的环境大都为 UNIX 的环境,因此,在规划、构建时,所花需的成本为最低,但 PHP 并未将 Presentation Layer 和 Business Layer 做一个适当的处理,因此,往往一个系统越来越庞大、越来越复杂时,维护起来就会越来越吃力,并且本身并没有一个强而有力的技术在支持它,当开发系统要求为分布式的体系结构时,那么 PHP 可能就英雄无用武之地了。

3-6 JSP 2.0 新功能

J2EE 1.4 正式发布之后,Servlet 和 JSP 同时也做了一些变动,Servlet 从 2.3 更新至 2.4;而 JSP 从 1.2 更新至 2.0。两者平心而论,JSP 的变动较 Servlet 来得多,其中 JSP 2.0 较 JSP 1.2 新增的功能如下:

- (1) Expression Language;
- (2) 新增 Simple Tag 和 Tag File;
- (3) *web.xml* 新增<jsp-config>元素。

3-6-1 Expression Language

JSP 2.0 之后,正式将 EL 纳入 JSP 的标准语法。EL 主要的功用在于简化 JSP 的语法,方便 Web 开发人员的使用。例如:

使用 JSP 传统语法:

```
<%
    String str_count = request.getParameter("count");
    int count = Integer.parseInt(str_count);
    count = count + 5;
    out.println("count: " + count);
%>
```

使用 EL 语法:

```
count: ${param.count + 5}
```

对于 EL 的部分,本书的“第六章: Expression Language”有详尽的介绍。

3-6-2 新增 Simple Tag 和 Tag File

JSP 2.0 提供一些较为简单的方法,让开发人员来撰写自定义标签。JSP 2.0 提供两种新的机制,分别为 Simple Tag 和 Tag File。

Simple Tag Handler 和其他 Tag Handler(如: Body Tag Handler、Tag Handler 和 Iteration Tag Handler)不同之处在于:Simple Tag Handler 并无 doStartTag()和 doEndTag(),它只有 doTag(),因此,实现标签能比以往更为方便。

JSP2.0 技术手册

Tag File 就更为简单，你可以把它当做直接使用 JSP 的语法来制作标签。例如：

■ Hello.tag

```
<%  
    out.println("Hello from tag file.");  
%>
```

我们先制作一个名为 *Hello.tag* 的 Tag File，然后将它放置在 *WEB-INF/tags/* 目录下。在 JSP 网页使用 *Hello.tag* 的方法如下：

```
<%@ taglib prefix="myTag" tagdir="/WEB-INF/tags" %>  
<myTag:Hello />
```

最后执行的结果如下：

Hello from tag file.

有关 Simple Tag Handler 和 Tag File 的部分，在“第十六章：Simple Tag 与 Tag File”有更详细的说明。

3-6-3 web.xml 新增<jsp-config>元素

~~<jsp-config> 元素主要用来设定 JSP 相关配置，<jsp-config> 包括 <taglib> 和 <jsp-property-group> 两个子元素。其中 <taglib> 元素在 JSP 1.2 时就已经存在；而 <jsp-property-group> 是 JSP 2.0 新增的元素。~~

<jsp-property-group> 元素主要有八个子元素，它们分别为：

<description>：设定的说明；

<display-name>：设定名称；

<url-pattern>：设定值所影响的范围，如：*/CH2* 或 */*.jsp*；

<el-ignored>：若为 true，表示不支持 EL 语法；

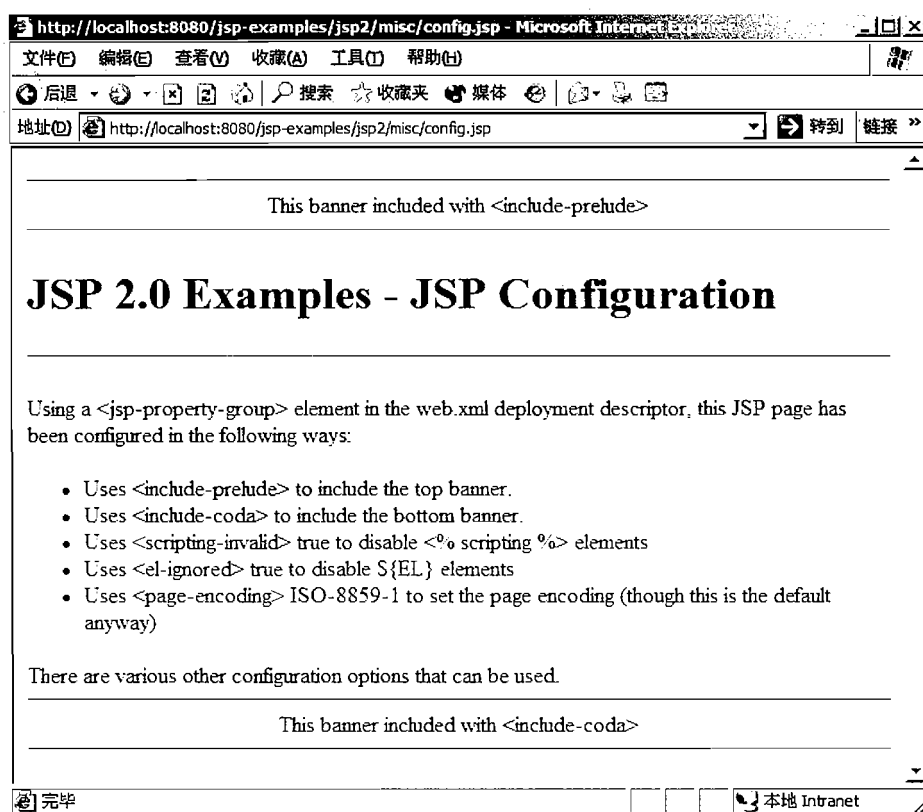
<scripting-invalid>：若为 true，表示不支持<% scripting %>语法；

<page-encoding>：设定 JSP 网页的编码；

<include-prelude>：设置 JSP 网页的抬头，扩展名为 *.jspf*；

<include-coda>：设置 JSP 网页的结尾，扩展名为 *.jspf*。

图3-3所示的所谓JSP网页的抬头为网页最上方的 This banner included with <include-prelude>；结尾为网页最下方的 This banner included with <include-coda>。

图 3-3 Tomcat 上的`<include-prelude>`范例程序

其中抬头的源程序为：

■ `prelude.jspf`

```
<hr>
<center>
This banner included with &lt;include-prelude&gt;
</center>
<hr>
```

结尾的源程序为：

■ `coda.jspf`

```
<hr>
<center>
This banner included with &lt;include-coda&gt;
</center>
<hr>
```

下面是一个简单的`<jsp-config>`元素完整配置：

JSP2.0 技术手册

```
<jsp-config>
  <taglib>
    <taglib-uri>Taglib</taglib-uri>
    <taglib-location>/WEB-INF/tlds/MyTaglib.tld</taglib-location>
  </taglib>

  <jsp-property-group>
    <description>
      Special property group for JSP Configuration JSP example.
    </description>
    <display-name>JSPConfiguration</display-name>
    <url-pattern>/jsp/* </url-pattern>
    <el-ignored>true</el-ignored>
    <page-encoding>GB2312</page-encoding>
    <scripting-invalid>true</scripting-invalid>
    <include-prelude>/include/prelude.jspf</include-prelude>
    <include-coda>/include/coda.jspf</include-coda>
  </jsp-property-group>
</jsp-config>
```


4

第四章

JSP 语法

前两章中，笔者已经对 Servlet 2.4 和 JSP 2.0 做了初步的介绍，本章主要介绍 JSP 的语法。4-1、4-2 两节算是了解 JSP 语法之前所必备的知识，请各位耐心看完。

本章将分为以下 7 节：

- 4-1 Elements 和 Template Data
- 4-2 批注(Comments)
- 4-3 Quoting 和 Escape 规则
- 4-4 Directives Elements
- 4-5 Scripting Elements
- 4-6 Action Elements
- 4-7 错误处理

JSP2.0 技术手册

4-1 Elements 和 Template Data

JSP 网页主要分为 Elements 与 Template Data 两部分。

Template Data: JSP Container 不处理的部分, 例如: HTML 的内容, 会直接送到 Client 端执行。

Elements: 必须经由 JSP Container 处理的部分, 而大部分 Elements 都以 XML 作为语法基础, 并且大小写必须要一致。

Elements 有两种表达式, 第一种为起始标签(包含 Element 名称、属性), 中间为一些内容, 最后为结尾标签。如下所示:

```
<mytag attr1="attribute value" ....>
  body
</mytag>
```

另一种是标签中只有 Element 的名称、属性, 称为 Empty Elements。如下所示:

```
<mytag attr1="attribute value" ..../>
```

Elements 有四种类型: Directive Elements、Scripting Elements、Action Elements 和 EL Elements, 接下来的章节会针对前三种类型的 Elements 加以说明。至于 EL Elements 是 JSP 2.0 新增的功能, 笔者将在“第六章: Expression Language”中详细介绍它。

4-2 批注 (Comments)

一般批注可分为两种: 一种为在客户端显示的批注; 另外一种就是客户端看不到, 只给开发程序员专用的批注。

● 客户端可以看到的批注:

```
<!-- comment [ <%= expression %> ] -->
```

例如:

```
<!-- 现在时间为: <%= (new java.util.Date()).toLocaleString() %> -->
```

在客户端的 HTML 源文件中显示为:

```
<!--现在时间为: January 1, 2004 -->
```

这种批注的方式和 HTML 中很像, 它可以使用“查看源代码”来看到这些程序代码, 但是惟有一些不同的是, 你可以在批注中加上动态的表达式(如上例所示)。

● 开发程序员专用的批注:

```
<%-- comment --%>
```

JSP2.0 技术手册

或者

```
<% /** this is a comment */ %>
```

接下来看下面这个范例:

```
<%@ page language="java" %>
<html>
<head><title>A Comment Test</title></head>
<body>
<h2>A Test of Comments</h2>
<%-- 这个批注不会显示在客户端 --%>
</body>
</html>
```

从用户的浏览器中,看到的源代码如下:

```
<html>
<head><title>A Comment Test</title></head>
<body>
<h2>A Test of Comments</h2>
</body>
</html>
```

之前加上去的批注在客户端的浏览器上看不出来,并且用此批注的方式,在 JSP 编译时会被忽略掉。这对隐藏或批注 JSP 程序是实用的方法,通常程序员也会利用它来调试(Debug)程序。

JSP Container 不会对 <%--和--%>之间的语句进行编译,它不会显示在客户端的浏览器上,也无法从源文件中看到。接下来介绍 Quoting 和 Escape 的规则。

4-3 Quoting 和 Escape 规则

Quoting 主要是为了避免与语法产生混淆所使用的转换功能,和 HTML 的标签语法类似,JSP 是以<%标签作为程序的起始、%>标签作为程序的结束,所以当你在 JSP 程序中要加上<%或是%>这些符号时,应该这么做:

■ Quoting.jsp

```
<%@ page contentType="text/html; charset=GB2312 " %>

<html>
<head>
  <title>CH4 - Quoting.jsp</title>
</head>
<body>

<h2>Quoting 范例程序</h2>

<%
  out.println("JSP 以%>作为结束符号");
```

JSP2.0 技术手册

```
%>

</body>
</html>
```

程序执行时, JSP 在执行到 %> 时, JSP Container 就直接告诉你程序有错误, 如图 4-1 所示:

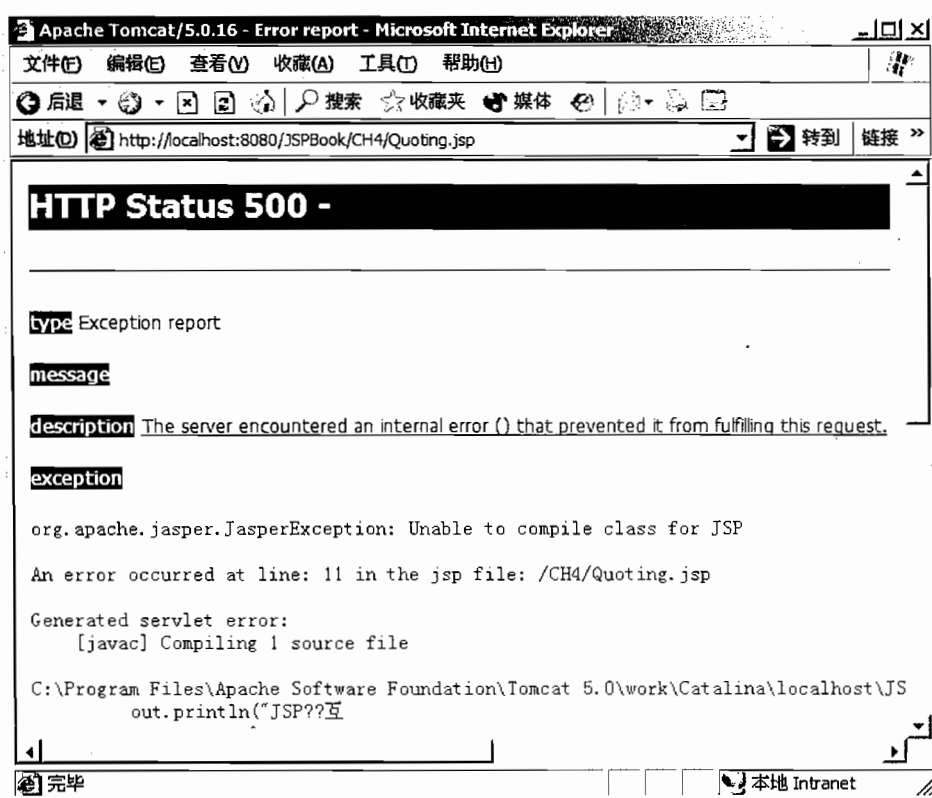


图 4-1 Quoting.jsp 的执行结果

通常为了避免产生这样的结果, 因此程序中遇到显示 %> 时, 要改写为 %\>, 所以上面的程序代码应改写为:

■ Quoting1.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>

<html>
<head>
  <title>CH4 - Quoting1.jsp</title>
</head>
<body>
```

```

<h2>Quoting 范例程序 2</h2>

<%
    out.println("JSP 以%\>作为结束符号");
%>

</body>
</html>

```

Quoting1.jsp 的执行结果如图 4-2 所示:

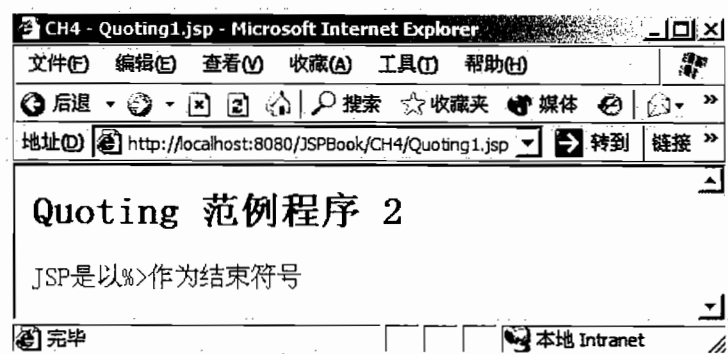


图 4-2 Quoting1.jsp 的执行结果

除了 %> 要改为 %\>, 当遇到<%、'、"、\ 时都要做适当修改, 如下所示:

单引号 ' 改为 \
 双引号 " 改为 \
 斜线 \ 改为 \
 起始标签 <% 改为 <%
 结束标签 %> 改为 %\>

最后再举个例子, 如下:

■ Quoting2.jsp

```

<%@ page contentType="text/html; charset=GB2312" %>

<html>
<head>
    <title>CH4 - Quoting2.jsp</title>
</head>
<body>

<h2>Quoting 范例程序 3</h2>

<%
    out.println("JSP 遇到 '\', '\", '\\, &lt;%, %\> 时需要做适当的修改");
%>

```

```
</body>  
</html>
```

执行的结果如图 4-3 所示:



图 4-3 Quoting2.jsp 的执行结果

4-4 Directives Elements

指令(Directives)主要用来提供整个 JSP 网页相关的信息, 并且用来设定 JSP 网页的相关属性, 例如: 网页的编码方式、语法、信息等。

起始符号为: `<%@`

终止符号为: `%>`

内文部分就是一些指令和一连串的属性设定, 如下所示:

```
<%@ directive { attribute = "value" } * %>
```

什么叫做一连串的属性设定? 举例来说, 当我们设定两个属性时, 可以将之合二为一, 如下:

```
<%@ directive attribute1 = "value1" %>  
<%@ directive attribute2 = "value2" %>
```

亦可以写成:

```
<%@ directive attribute1 = "value1" attribute2 = "value2" %>
```

在 JSP 1.2 的规范中, 有三种指令: `page`、`include` 和 `taglib`, 每一种指令都有各自的属性。接下来, 我们会依序为各位读者介绍这三种指令。

JSP 2.0 新增 Tag File 的功能, Tag File 是以 `.tag` 作为扩展名的。在 Tag File 中, 因为它并不是 JSP 网页, 所以不能使用 `page` 指令, 但是它可以使用 `include` 和 `taglib` 指令。除此之外, Tag File 还有自己本身的指令可以使用, 如: `tag`、`attribute` 和 `variable`。

有关 Tag File 的指令, 笔者在“第十六章: Simple Tag 与 Tag File”中再做详细介绍, 本章

JSP2.0 技术手册

先暂时不谈论它。

4-4-1 page 指令

page 指令是最复杂的 JSP 指令, 它的主要功能为设定整个 JSP 网页的属性和相关功能。page 指令的基本语法如下:

```
<%@ page attribute1="value1" attribute2= "value2" attribute3=...%>
```

page 指令是以<%@ page 起始, 以%>结束。像所有 JSP 标签元素一样, page 指令也支持以 XML 为基础的语法, 如下所示:

```
<jsp:directive.page attribute1="value1" attribute2= "value2" />
```

page 指令有 11 个属性, 如表 4-1 所示:

表 4-1

属 性	定 义
language = "scriptingLanguage"	主要指定 JSP Container 要使用什么语言来编译 JSP 网页。JSP 2.0 规范中指出, 目前只可以使用 Java 语言, 不过未来不排除增加其他语言, 如 C、C++、Perl 等等。默认值为 Java
extends = "className"	主要定义此 JSP 网页产生的 Servlet 是继承哪个父类
import = "importList"	主要定义此 JSP 网页可以使用哪些 Java API
session = "true false"	决定此 JSP 网页是否可以使用 session 对象。默认值为 true
buffer = "none size in kb"	决定输出流 (output stream) 是否有缓冲区。默认值为 8KB 的缓冲区
autoFlush = "true false"	决定输出流的缓冲区是否要自动清除, 缓冲区满了会产生异常 (Exception)。默认值为 true
isThreadSafe = "true false"	主要是告诉 JSP Container, 此 JSP 网页能处理超过一个以上的请求。默认值为 true, 如果此值设为 false, SingleThreadModel 将会被使用。SingleThreadModel 在 Servlet 2.4 中已经声明不赞成使用 (deprecate)
info = "text"	主要表示此 JSP 网页的相关信息
errorPage = "error_url"	表示如果发生异常错误时, 网页会被重新指向那一个 URL
isErrorPage = "true false"	表示此 JSP Page 是否为处理异常错误的网页
contentType = "ctinfo"	表示 MIME 类型和 JSP 网页的编码方式
pageEncoding = "ctinfo"	表示 JSP 网页的编码方式
isELIgnored = "true false"	表示是否在此 JSP 网页中执行或忽略 EL 表达式。如果为 true 时, JSP Container 将忽略 EL 表达式; 反之为 false 时, EL 表达式将会被执行

下面的范例都合乎语法规则:

JSP2.0 技术手册


```
<%@ page info = "this is a jsp page"%>
<%@ page language = "java" import = "java.net.* " %>
<%@ page import = "java.net.*,java.util.List " %>
```

下面的范例也是 page 指令，不过并不合乎语法规则，因为 session 属性重复设定两次：

```
<%@ page language = "java" import = "java.net.* " session = "false" buffer =
    "16kb" autoFlush = "false" session = "false" %>
```

注意：只有 import 这个属性可以重复设定，其他则否。

```
<%@ page import = "java.net.* " %>
<%@ page import = "java.util.List " %>
```

另外再举个较常见的错误例子：

```
<%@ page language="java" contentType="text/html";charset = "Big5" %>
```

应该改为：

```
<%@ page language="java" contentType="text/html; charset=Big5" %>
```

通常我们都以为只要把 charset 设为 Big5 的编码方式，就能够顺利显示出所需要的中文，不过有时候在显示一些特别的中文字时，例如：碁，会变成乱码。如下范例：

■ Big5.jsp

```
<%@ page contentType="text/html; charset=Big5" %>
```

```
<html>
<head>
    <title>CH4 - Big5.jsp</title>
</head>
<body>
    <h2>使用 Big5 編碼，無法正確顯示某些中文字</h2>
```

```
<%
    out.println("宏?碁q脑公司");
%>
```

```
</body>
</html>
```

Big5.jsp 的执行结果如图 4-4 所示：

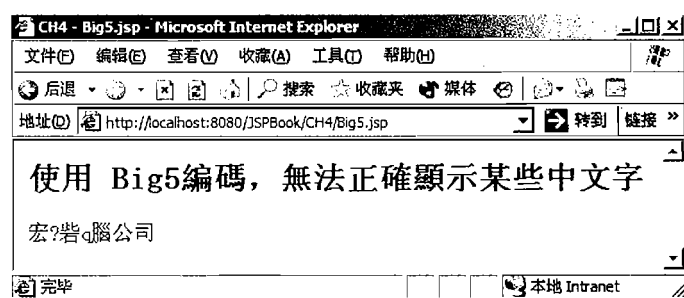


图 4-4 Big5.jsp 的执行结果

那么我们应该如何解决这个问题？很简单，只要把之前的 `charset = Big5` 改为 `charset = MS950` 的编码方式就能够解决这个问题，我们看下面这个范例：*MS950.jsp*。

■ MS950.jsp

```
<%@ page contentType="text/html; charset=MS950" %>

<html>
<head>
  <title>CH4 - MS950.jsp</title>
</head>
<body>

<h2>使用 MS950 編碼，能正確顯示"碁"</h2>

<%
    out.println("宏碁電腦公司");
%>

</body>
</html>
```

MS950.jsp 的执行结果如图 4-5:

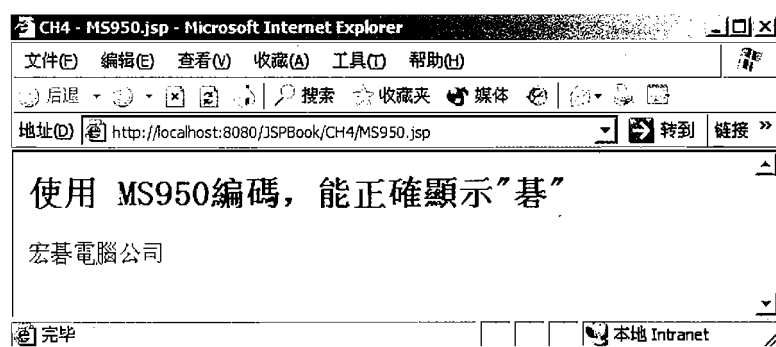


图 4-5 MS950.jsp 的执行结果

使用最基本的 page 指令的范例程序 (二): *Date.jsp*

■ *Date.jsp*

```
<%@ page contentType="text/html; charset=GB2312 " %>
<%@ page import="java.util.Date" %>

<html>
<head>
  <title>CH4 - Date.jsp</title>
</head>
<body>

<h2>使用 java.util.Date 显示目前时间</h2>

<%
  Date date = new Date();
  out.println("现在时间: "+date);
%>

</body>
</html>
```

因为 *Date.jsp* 要显示出现在的时间, 所以要先导入(import) *java.util* 这个套件, 才可以使用 *Date()* 类。执行结果如图 4-6 所示:



图 4-6 *Date.jsp* 的执行结果

如果在一个 JSP 网页同时须要导入很多套件时:

```
<%@ page import="java.util.Date" %>
<%@ page import="java.text.*" %>
```

亦可以写为:

```
<%@ page import="java.util.Date, java.text.*" %>
```

直接使用逗号 “,” 分开, 然后就可以一直串接下去。

4-4-2 include 指令

include 指令表示：在 JSP 编译时插入一个包含文本或代码的文件，这个包含的过程是静态的，而包含的文件可以是 JSP 网页、HTML 网页、文本文件，或是一段 Java 程序。

注意

包含文件中要避免使用<html>、</html>、<body>、</body>，因为这将会影响在原来 JSP 网页中同样的标签，这样做有时会导致错误。

include 指令的语法如下：

```
<%@ include file = "relativeURLspec" %>
```

include 指令只有一个属性，那就是 file，而 relativeURLspec 表示此 file 的路径。像所有 JSP 标签元素一样，include 指令也支持以 XML 为基础的语法，如下所示：

```
<jsp:directive.include file = "relativeURLspec" />
```

注意

<%@ include %>指令是静态包含其他的文件。所谓的静态是指 file 不能为一变量 URL，例如：

```
<% String URL="JSP.html" ; %>
<%@ include file = "<%= URL %>" %>
```

也不可以在 file 所指定的文件后接任何参数，如下：

```
<%@ include file = "javaworld.jsp?name=browser" %>
```

同时，file 所指的路径必须是相对于此 JSP 网页的路径。

笔者写了一个 *Include.jsp* 范例程序，它 include 一份 HTML 的网页，网页文件名叫做 *Hello.html*，看看执行之后，会有什么结果产生。

■ Include.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>

<html>
<head>
  <title>CH4 - Include_Html.jsp</title>
</head>
<body>

  <h2>include 指令</h2>

  <%@ include file="Hello.html" %>
  <%
    out.println("欢迎大家进入 JSP 的世界");
  %>
```

JSP2.0 技术手册

```
</body>
</html>
```

■ *Hello.html*
JSP 2.0 Tech Reference

执行结果如图 4-7。

注意
Hello.html 网页内容中有中文时，Tomcat 5.0.16 执行 Include.jsp 时，无法正确显示 Hello.html 网页的中文，会产生乱码。但是笔者使用 Resin 来执行时，却可以顺利显示中文。



图 4-7 Include.jsp 的执行结果

4-4-3 taglib 指令

taglib 指令是 JSP 1.1 新增进来的功能，能够让用户自定义新的标签。这里只是先做一个简单介绍，在第十五章再为各位读者详细介绍。

taglib 指令的语法如下：

```
<%@ taglib uri = "tagLibraryURI" prefix="tagPrefix" %>
```

像所有 JSP 标签元素一样，taglib 指令也支持以 XML 为基础的语法，如下所示（见表 4-2）：

```
<jsp:directive.taglib uri = "tagLibraryURI" prefix="tagPrefix" />
```

表 4-2

属 性	定 义
uri = "tagLibraryURI"	主要是说明 taglibrary 的存放位置
prefix="tagPrefix"	主要用来区分多个自定义标签

范例：

```
<%@ taglib uri = "/supertags/" prefix="super" %>
.....
```

```
<super:doMagic>
    .....
    .....
</super:doMagic>
```

4-5 Scripting Elements

Scripting Elements 包含三部分:

1. 声明(Declarations)
2. Scriptlets
3. 表达式 (Expressions)

示例:

```
<%!      这是声明      %>
<%      这是 Scriptlets  %>
<%=      这是表达式      %>
```

4-5-1 声明 (Declarations)

在 JSP 程序中声明合法的变量和方法。声明是以<%! 为起始; %> 为结尾。

声明的语法:

```
<%! declaration; [ declaration; ]+ ... %>
```

范例:

```
<%! int i = 0; %>
<%! int a, b, c; %>
<%! Circle a = new Circle(2.0); %>
<%! public String f(int i) {if ( i < 3 ) return ("i 小于3");} %>
```

使用<%! %>可以声明你在 JSP 程序中要用的变量和方法, 你可以一次声明多个变量和方法, 只要最后以分号“;”结尾就行, 当然这些声明在 Java 中要是合法的。

每一个声明仅在一个页面中有效, 如果你想每个页面都用到一些声明, 最好把它们写成一个单独的 JSP 网页, 然后用<%@ include %>或<jsp:include >元素包含进来。

注意

使用<%! %>方式所声明的变量为全局变量, 即表示: 若同时 n 个用户在执行此 JSP 网页时, 将会共享此变量。因此笔者强烈建议读者, 千万别使用<%! %>来声明您的变量。

若要声明变量时, 请直接在<% %>之中声明使用即可。

4-5-2 Scriptlets

Scriptlet 中可以包含有效的程序片段，只要是合乎 Java 本身的标准语法即可。通常我们主要的程序也是写在这里面，Scriptlet 是以 `<%` 为起始；`%>` 为结尾。

Scriptlet 的语法：

```
<% code fragment %>
```

范例：

```
<%
    String name = null;
    if (request.getParameter("name") == null) {
%>
.....
.....
<%
    }
    else
    {
        out.println("HI ... "+name);
    }
%>
```

Scriptlet 能够包含多个语句，方法，变量，表达式，因此它能做以下的事：

- (1) 声明将要用到的变量或方法；
- (2) 显示出表达式；
- (3) 使用任何隐含的对象和使用 `<jsp:useBean>` 声明过的对象，编写 JSP 语句（如果你在使用 Java 语言，这些语句必须遵从 Java Language Specification）；
- (4) 当 JSP 收到客户端的请求时，Scriptlet 就会被执行，如同 Servlet 的 `doGet()`、`doPost()`，如果 Scriptlet 有显示的内容会被存在 `out` 对象中，然后再利用 `out` 对象中的 `println()` 方法显示出结果。

4-5-3 表达式 (Expressions)

Expressions 标签是以 `<%=` 为起始；`%>` 为结尾，其中间内容包含一段合法的表达式，例如：使用 Java 的表达式。

Expressions 的语法：

```
<%= expression %>
```

范例：

```
<font color="blue"><%= getName() %></font>
<%= (new java.util.Date()).toLocaleString() %>
```

表达式在执行后会被自动转化为字符串，然后显示出来。

当你在 JSP 中使用表达式时请记住以下几点:

1. 不能使用分号 “;” 来作为表达式的结束符号, 如下:

```
<%= (new java.util.Date()).toLocaleString(); %>
```

但是同样的表达式用在 Scriptlet 中就需要以分号来结尾了。

2. 这个表达式元素能够包括任何 Java 语法, 有时候也能作为其他 JSP 元素的属性值。

4-6 Action Elements

JSP 2.0 规范中定义一些标准 action 的类型, JSP Container 在实现时, 也完全遵照这个规范而制定。Action 元素的语法以 XML 为基础, 所以, 在使用时大小写是有差别的, 例如: `<jsp:getproperty>` 和 `<jsp:getProperty>` 是有所差别的, 因此在撰写程序时, 必须要特别注意。

目前 JSP 2.0 规范中, 主要有 20 项 Action 元素:

```
<jsp:useBean>
<jsp:setProperty>
<jsp:getProperty>

<jsp:include>
<jsp:forward>
<jsp:param>
<jsp:plugin>
<jsp:params>
<jsp:fallback>

<jsp:root>
<jsp:declaration>
<jsp:scriptlet>
<jsp:expression>
<jsp:text>
<jsp:output>

<jsp:attribute>
<jsp:body>
<jsp:element>

<jsp:invoke>
<jsp:doBody>
```

笔者将这 20 个 action 元素分为五类:

第一类有 3 个 action 元素, 它们都用来存取 JavaBean, 因此这部分将在“第八章: JSP 与

JavaBean”详细地介绍。

第二类有 6 个 action 元素，这部分是 JSP 1.2 原有的 action 元素，接下来将会介绍它们。

第三类有 6 个 action 元素，它们主要用在 JSP Document 之中。其中<jsp:output>是 JSP 2.0 新增的元素。

第四类有 3 个 action 元素，它们主要用来动态产生 XML 元素标签的值，这 3 个都是在 JSP 2.0 中加入进来的元素。

第五类有 2 个 action 元素，它们主要用在 Tag File 中，这部分将在“第十六章：Simple Tag 与 Tag File”再来介绍。

补充

JSP Document: 使用 XML 语法所写成的 JSP 网页。例如:

```
<jsp:scriptlet>
    String name="Mike";
</jsp:scriptlet>
Hi !<jsp:expression>name</jsp:expression>
```

4-6-1 <jsp:include>

<jsp:include>元素允许你包含动态和静态文件，这两种产生的结果是不尽相同的。如果包含进来的只是静态文件，那么只是把静态文件的内容加到 JSP 网页中；如果包含进来的为动态文件，那么这个被包含的文件也会被 JSP Container 编译执行。

一般而言，你不能直接从文件名称上来判断一个文件是动态的还是静态的，例如：*Hello.jsp*就有可能只是单纯包含一些信息而已，而不须要执行。但是<jsp:include>能够自行判断此文件是动态的还是静态的，于是能同时处理这两种文件。

<jsp:include>的语法:

```
<jsp:include page="{urlSpec | <%= expression %>}" flush="true | false" />
```

或

```
<jsp:include page="{urlSpec | <%= expression %>}" flush="true | false" >
    <jsp:param name="PN" value="{PV | <%= expression %>}" /> *
</jsp:include>
```

说明:

<jsp:include>有两个属性: page 和 flush。

page: 可以代表一个相对路径, 即你所要包含进来的文件位置或是经过表达式所运算出的相对路径。

flush: 接受的值为 boolean, 假若为 true, 缓冲区满时, 将会被清空。flush 的默认值为 false。

在此需要补充一点: 在 JSP 1.2 之前, flush 必须设为 true。

你还可以用<jsp:param>传递一个或多个参数给 JSP 网页。

JSP2.0 技术手册

范例:

```
<jsp:include page="scripts/Hello.jsp" />
<jsp:include page="Hello.html" />
<jsp:include page="scripts/login.jsp">
    <jsp:param name="username" value="browser" />
    <jsp:param name="password" value="1234" />
</jsp:include>
```

4-6-2 <jsp:forward>

<jsp:forward>这个标签的定义: 将客户端所发出来的请求, 从一个 JSP 网页转交给另一个 JSP 网页。不过有一点要特别注意, <jsp:forward>标签之后的程序将不能执行。笔者用例子来说明:

```
<%
    out.println("会被执行 !!!");
%>
<jsp:forward page="Quoting2.jsp">
<jsp:param name="username" value="Mike" />
</jsp:forward>

<%
    out.println("不会执行 !!!");
%>
```

上面这个范例在执行时, 会打印出“会被执行 !!!”, 不过随后马上会转入到 *SayHello.jsp* 的网页中, 至于 `out.println("不会执行 !!!")` 将不会被执行。

<jsp:forward>的语法:

```
<jsp:forward page="{relativeURL" | "<%= expression %>"}" />
```

或

```
<jsp:forward page="{relativeURL" | "<%= expression %>"}" >
    <jsp:param name="PN" value="{PV | <%= expression %>}" /> *
</jsp:forward>
```

说明:

如果你加上<jsp:param>标签, 你就能够向目标文件传送参数和值, 不过这些目标文件必须也是一个能够取得这些请求参数的动态文件, 例如: *.cgi*、*.php*、*.asp* 等等。

<jsp:forward>只有一个属性 `page`。page 的值, 可以是一个相对路径, 即你所要重新导向的网页位置, 亦可以是经过表达式运算出的相对路径。

范例:

```
<jsp:forward page="/SayHello.jsp" />
```

或者

```
<jsp:forward page="/SayHello.jsp">
  <jsp:param name="username" value="Mike" />
</jsp:forward>
```

4-6-3 <jsp:param>

<jsp:param>用来提供 key/value 的信息，它也可以与<jsp:include>、<jsp:forward>和<jsp:plugin> 一起搭配使用。

当在用<jsp:include>或者<jsp:forward>时，被包含的网页或转向后的网页会先看看 request 对象里除了原本的参数值之外，有没有再增加新的参数值，如果有增加新的参数值时，则新的参数值在执行时，有较高的优先权。例如：

一个 request 对象有一个参数 A = foo；另一个参数 A = bar 是在转向时所传递的参数，则网页中的 request 应该会为 A = bar,foo。注意：新的参数值有较高的优先权。

<jsp:param>的语法：

```
<jsp:param name="ParameterName" value="ParameterValue" />
```

<jsp:param>有两个属性：name 和 value。name 的值就是 parameter 的名称；而 value 的值就是 parameter 的值。

范例：

```
<jsp:param name="username" value="Mike" />
<jsp:param name="password" value="Mike007" />
```

4-6-4 <jsp:plugin>、<jsp:params>和<jsp:fallback>

<jsp:plugin>用于在浏览器中播放或显示一个对象(通常为 Applet 或 Bean)。

当 JSP 网页被编译后送往浏览器执行时，<jsp:plugin>将会根据浏览器的版本替换成<object>标签或者<embed>标签。一般来说，<jsp:plugin>会指定对象 Applet 或 Bean，同样也会指定类的名字和位置，另外还会指定将从哪里下载这个 Java 组件。

注意
<object>用于 HTML 4.0，<embed>用于 HTML 3.2。

<jsp:plugin>的语法：

```
<jsp:plugin type="bean | applet"
  code="objectCode"
  codebase="objectCodebase"
  [ align="alignment" ]
  [ archive="archiveList" ]
  [ height="height" ]
  [ hspace="hspace" ]
  [ jreversion="jreversion" ]
```

JSP2.0 技术手册

```

        [ name="ComponentName" ]
        [ vspace="vspace" ]
        [ width="width" ]
        [ nspluginurl="URL" ]
        [ iepluginurl="URL" ] >
[ <jsp:params>
  [ <jsp:param name="PN" value="{PV | <%= expression %>}" /> ] +
  </jsp:params> ]

[ <jsp:fallback> text message for user </jsp:fallback> ]
</jsp:plugin>

```

说明:● **type="bean | applet":**

对将被执行的对象类型，你必须指定是 Bean 还是 Applet，因为这个属性没有默认值。

● **code="objectCode":**

将被 Java Plugin 执行的 Java 类名称，必须以 **.class** 结尾，并且 **.class** 类文件必须存在于 codebase 属性所指定的目录中。

● **codebase="objectCodebase":**

如果你没有设定将被执行的 Java 类的目录（或者是路径）的属性，默认值为使用 <jsp:plugin> 的 JSP 网页所在目录。

● **align="alignment" :**

图形、对象、Applet 的位置。align 的值可以为：
bottom、top、middle、left、right

● **archive=" archiveList":**

一些由逗号分开的路径名用于预先加载一些将要使用的类，此做法可以提高 Applet 的性能。

● **name=" ComponentName":**

表示这个 Bean 或 Applet 的名字。

● **height="height" width="width":**

显示 Applet 或 Bean 的长、宽的值，单位为像素（pixel）。

● **hspace="hspace" vspace="vspace":**

表示 Applet 或 Bean 显示时在屏幕左右、上下所需留下的空间，单位为像素（pixel）。

● **jreversion="jreversion":**

表示 Applet 或 Bean 执行时所需的 Java Runtime Environment (JRE) 版本，默认值是 1.1。

● **nspluginurl="URL":**

表示 Netscape Navigator 用户能够使用的 JRE 的下载地址，此值为一个标准的 URL。

● **iepluginurl="URL":**

表示 IE 用户能够使用的 JRE 的下载地址，此值为一个标准的 URL。

JSP2.0 技术手册

● `<jsp:params>`

```
[ <jsp:param name="PN" value="{PV | <%= expression %>}" /> ] +
</jsp:params>
```

你可以传送参数给 Applet 或 Bean。

● `<jsp:fallback> unable to start plugin </jsp:fallback>`

一段文字用于：当不能启动 Applet 或 Bean 时，那么浏览器会有一段错误信息。

范例：

```
<jsp:plugin type="applet" code="Molecule.class" codebase="/html">
  <jsp:params>
    <jsp:param name="molecule" value="molecules/benzene.mol" />
  </jsp:params>
  <jsp:fallback>
    <p>Unable to start plugin</p>
  </jsp:fallback>
</jsp:plugin>
```

4-6-5 `<jsp:element>`、`<jsp:attribute>`和`<jsp:body>`

`<jsp:element>`元素用来动态定义 XML 元素标签的值。

`<jsp:element>`的语法：

```
<jsp:element name="name">
  本体内内容
</jsp:element>
```

或

```
<jsp:element name="name">
  <jsp:attribute>
  ...
</jsp:attribute>
  ...
  <jsp:body>
  ...
</jsp:body>
</jsp:element>
```

`<jsp:element>`只有一个属性 `name`。`name` 的值就是 XML 元素标签的名称。

范例 1：

```
<jsp:element name="firstname"></jsp:element>
```

执行的结果如下：

```
<firstname></firstname>
```

范例 2：


```
<jsp:element name="firstname">
  <jsp:attribute name="name">Mike</jsp:attribute>
  <jsp:body>Hello</jsp:body>
</jsp:element>
```

执行的结果如下:

```
<firstname name="Mike">Hello</firstname>
```

4-6-6 <jsp:attribute>

<jsp:attribute>元素主要有两个用途:

- (1) 当使用在<jsp:element>之中时, 它可以定义 XML 元素的属性, 如上述的范例 2。
- (2) 它可以用来设定标准或自定义标签的属性值。如下范例 1:

<jsp:attribute>的语法:

```
<jsp:attribute name="name" trim="true | false">
  本内容
</jsp:attribute >
```

<jsp:attribute>有两个属性: name 和 trim。其中 name 的值就是标签的属性名称。trim 可为 true 或 false。假若为 true 时, <jsp:attribute>本内容的前后空白, 将被忽略; 反之, 若为 false, 前后空白将不被忽略。trim 的默认值为 true。

范例:

```
<jsp:useBean id="foo" class="jsp2.examples.FooBean">
  Bean created! Setting foo.bar...<br>
  <jsp:setProperty name="foo" property="bar">
    <jsp:attribute name="value">
      Hello World
    </jsp:attribute>
  </jsp:setProperty>
</jsp:useBean>
<br>
Result: <jsp:getProperty name="foo" property="bar">
```

执行的结果如下:

```
Bean created! Setting foo.bar...
Result: Hello World
```

其实上述的范例和下面的例子一样:

```
<jsp:useBean id="foo" class="jsp2.examples.FooBean">
  Bean created! Setting foo.bar...<br>
  <jsp:setProperty name="foo" property="bar" value="Hello World" >
  </jsp:setProperty>
</jsp:useBean>
<br>
Result: <jsp:getProperty name="foo" property="bar">
```


有关<jsp:useBean>、<jsp:setProperty>和<jsp:getProperty>在“第八章:JSP 与 JavaBean”会有更详细的说明。

4-6-7 <jsp:body>

<jsp:body>用来定义 XML 元素标签的本体内容。

<jsp:body>的语法:

```
<jsp:body>
  本体内容
</jsp:body>
```

<jsp:body>没有任何的属性。

范例 1:

```
<jsp:element name="firstname">
  <jsp:attribute name="name">Mike</jsp:attribute>
  <jsp:body>Hello</jsp:body>
</jsp:element>
```

执行的结果如下:

```
<firstname name="Mike">Hello</firstname>
```

范例 2:

```
<jsp:element name="firstname">
  <jsp:attribute name="name">Mike</jsp:attribute>
</jsp:element>
```

执行的结果如下:

```
<firstname name="Mike" />
```

4-7 错误处理

通常 JSP 在执行时,在两个阶段下会发生错误。

- JSP 网页 → Servlet 类
- Servlet 类处理每一个请求时

在第一阶段时,产生的错误我们称为 Translation Time Processing Errors;在第二阶段时,产生的错误我们称为 Client Request Time Processing Errors。接下来我们将针对这两个阶段产生错误的原因和处理方法,为各位读者做详细的介绍。

4-7-1 Translation Time Processing Errors

Translation Time Processing Errors 产生的主要原因：我们在撰写 JSP 时的语法有错误，导致 JSP Container 无法将 JSP 网页编译成 Servlet 类文件(.class)，例如：500 Internal Server Error，500 是指 HTTP 的错误状态码，因此是 Server Error，如图 4-8。

通常产生这种错误时，可能是 JSP 的语法有错误，或是 JSP Container 在一开始安装、设定时，有不适当的情形发生。解决的方法就是再一次检查程序是否有写错的，不然也有可能是 JSP Container 的 bug。

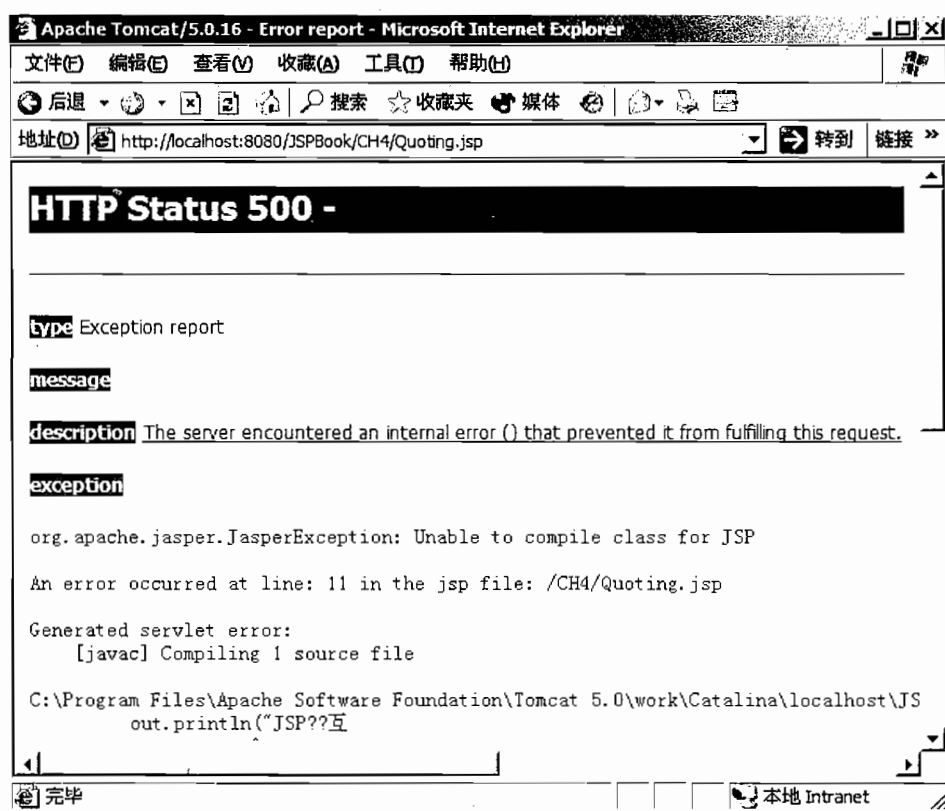


图 4-8 500 Internal Server Error

4-7-2 Client Request Time Processing Errors

Client Request Time Processing Errors 错误的发生，往往不是语法错误，而可能是逻辑上的错误，简单地说，你写一个计算除法的程序，当用户输入的分母为零时，程序会发生错误并抛出异常(Exception)，交由异常处理(Exception Handling)机制做适当的处理。

对于这种错误的处理，我们通常会交给 `errorPage` 去处理。下面举个例子，我想各位读者

JSP2.0 技术手册

应该能够很容易了解。

使用 `errorPage` 的范例程序 (一): `ErrorPage.jsp`

■ `ErrorPage.jsp`

```
<%@ page contentType="text/html; charset=GB2312" errorPage="Error.jsp" %>

<html>
<head>
  <title>CH4 - ErrorPage.jsp</title>
</head>
<body>

<h2>errorPage 的范例程序</h2>

<%!
  private double toDouble(String value)
  {
    return(Double.valueOf(value).doubleValue());
  }
%>
<%
  double num1 = toDouble(request.getParameter("num1"));
  double num2 = toDouble(request.getParameter("num2"));
%>
您传入的两个数字为: <%= num1 %> 和 <%= num2 %><br>
两数相加为 <%= (num1+num2) %>

</body>
</html>
```

`ErrorPage.jsp` 程序中, 我们使用 `page` 指令中的 `errorPage` 属性, 告诉 JSP Container, 如果在程序中有错误产生时, 会自动交给 `Error.jsp` 处理。

```
<%@ page contentType="text/html; charset=GB2312 " errorPage="Error.jsp" %>
```

我们声明一个函数叫 `toDouble`, 主要把我们接收进来的字符串转成 `Double` 的类型, 才能做加法运算。

```
<%!
  private double toDouble(String value)
  {
    return(Double.valueOf(value).doubleValue());
  }
%>
```

接收 `num1` 和 `num2` 两个字符串, 并且将它们转为 `double` 的类型, 并各自命名为 `num1` 和 `num2`。

```
<%
  double num1 = toDouble(request.getParameter("num1"));
  double num2 = toDouble(request.getParameter("num2"));
%>
```

JSP2.0 技术手册

■ Error.jsp

```
<%@ page contentType="text/html; charset=GB2312" isErrorPage="true" %>
<%@ page import="java.io.PrintWriter" %>

<html>
<head>
  <title>CH4 - Error.jsp</title>
</head>
<body>

<h2>errorPage 的范例程序</h2>

<p>ErrorPage.jsp 错误产生: <I><%= exception %></I></p><br>
<pre>
问题如下: <% exception.printStackTrace(new PrintWriter(out)); %>
</pre>

</body>
</html>
```

Error.jsp 主要处理 *ErrorPage.jsp* 所产生的错误, 所以在 *ErrorPage.jsp* 中 *page* 指令的属性 *errorPage* 设为 *Error.jsp*, 因此, 若 *ErrorPage.jsp* 有错误发生时, 会自动转到 *Error.jsp* 来处理。

Error.jsp 必须设定 *page* 指令的属性 *isErrorPage* 为 *true*, 因为 *Error.jsp* 是专门用来处理错误的网页。

```
<%@ page contentType="text/html; charset=GB2312" isErrorPage="true" %>
```

下面这几行代码用来显示出错误发生的原因:

```
<p>ErrorPage.jsp 错误产生: <I><%= exception %></I></p><br>
<PRE>
问题如下: <% exception.printStackTrace(new PrintWriter(out)); %>
</PRE>
```

由于在这个程序中并没有做一个窗体来输入两个数字, 所以必须手动在 URL 后输入 *num1* 和 *num2* 的值。如图 4-9 所示:



图 4-9 ErrorPage.jsp 的执行结果

上一个范例的参数设定为:

`http://localhost:8080/JSPBook/CH4/ErrorMessage.jsp?num1=100&num2=245`

此时 `num1` 的值为 100, `num2` 的值为 245, 所以相加的结果就为 345。但是, 如果 `num1` 或 `num2` 的参数不是数字的话, 例如:

`http://localhost:8080/JSPBook/CH4/ErrorMessage.jsp?num1=javaWorld&num2=245`

那么就会产生错误结果如图 4-10。

当 `ErrorMessage.jsp` 产生错误时, 就会交由 `Error.jsp` 去处理, 所以您看到的结果, 其实是执行 `Error.jsp` 后的结果。



图 4-10 `ErrorMessage.jsp` 执行时产生错误

5

第五章

隐含对象 (Implicit Object)

所谓隐含对象(Implicit Object)，就是当你在撰写 JSP 网页时，不须做任何
的声明(declare)就可以直接使用的对象。

下面将分 5 节来为读者介绍：

- 5-1 属性(Attribute)与范围(Scope)
- 5-2 与 Servlet 有关的隐含对象
- 5-3 与 Input / Output 有关的隐含对象
- 5-4 与 Context 有关的隐含对象
- 5-5 与 Error 有关的隐含对象

JSP2.0 技术手册

表 5-1 列出了目前在 JSP 2.0 规范中所定义的九个隐含对象：

表 5-1

隐含对象	类 型	说 明
request	javax.servlet.http.HttpServletRequest	请求端信息
response	javax.servlet.http.HttpServletResponse	响应端信息
pageContext	javax.servlet.jsp.PageContext	表示此 JSP 的 PageContext
session	javax.servlet.http.HttpSession	在同联机中, 所产生的 session 数据, 目前只对 HTTP 协议有意义
application	javax.servlet.ServletContext	如同调用 <code>getServletConfig().getServletContext()</code>
out	javax.servlet.jsp.JspWriter	数据流的标准输出
config	javax.servlet.ServletConfig	表示此 JSP 的 ServletConfig
page	java.lang.Object	如同 Java 的 this
exception	java.lang.Throwable	异常处理

这九个隐含对象, 笔者将它们分为四大类：

1. 与 Servlet 有关的隐含对象

- page
- config

2. 与 Input / Output 有关的隐含对象

- out
- request
- response

3. JSP 执行时, 提供有关 Context 的隐含对象

- session
- application
- pageContext

4. 与 Error 有关的隐含对象

- exception

5-1 属性(Attribute)与范围(Scope)

有些 JSP 程序员会将 request、session、application 和 pageContext 归为一类,原因在于:它们皆能借助 setAttribute()和 getAttribute()来设定和取得其属性(Attribute),通过这两种方法来做到数据分享。

我们先来看下面这段小程序:

■ Page1.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>

<html>
<head>
  <title>CH5 - Page1.jsp</title>
</head>
<body>

</br>
<%
  application.setAttribute("Name", "mike");
  application.setAttribute("Password", "browser");
%>
<jsp:forward page="Page2.jsp"/>

</body>
</html>
```

在这个程序中,笔者设定两个属性: Name、Password, 其值为: mike、browser。然后再转交(forward)到 Page2.jsp。我只要在 Page2.jsp 当中加入 application.getAttribute(), 就能取得在 Page1.jsp 设定的数据。

■ Page2.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>

<html>
<head>
  <title>CH5 - Page2.jsp</title>
</head>
<body>

<%
  String Name = (String) application.getAttribute("Name");
  String Password = (String) application.getAttribute("Password");
  out.println("Name = " + Name);
  out.println("Password = " + Password);
%>

</body>
```

JSP2.0 技术手册

```
</html>
```

Page1.jsp 的执行结果如图 5-1 所示。



图 5-1 *Page1.jsp* 的执行结果

从图 5-1 可以看出,此时 Name 的值就会等于 mike, Password 的值就等于 browser。看完这个小范例之后,读者有没有发现网页之间要传递数据时,除了可以使用窗体、隐藏字段来完成之外, JSP 技术还提供给开发人员一项传递数据的机制,那就是利用 `setAttribute()` 和 `getAttribute()` 方法,如同 *Page1.jsp* 和 *Page2.jsp* 的做法。不过它还是有些限制的,这就留到下一节来说明。

在上面 *Page1.jsp* 和 *Page2.jsp* 的程序当中,是将数据存入到 `application` 对象之中。除了 `application` 之外,还有 `request`、`pageContext` 和 `session`,也都可以设定和取得属性值,那它们之间有什么分别吗?

它们之间最大的差别在于范围(Scope)不一样,这个概念有点像 C、C++ 中的全局变量和局部变量的概念。接下来就介绍 JSP 的范围。

5-1-1 JSP Scope — Page

JSP 有四种范围,分别为 Page、Request、Session、Application。所谓的 Page,指的是单页 JSP Page 的范围。若要将数据存入 Page 范围时,可以用 `pageContext` 对象的 `setAttribute()` 方法;若要取得 Page 范围的数据时,可以使用 `pageContext` 对象的 `getAttribute()` 方法。我们将之前的范例做小幅度的修改,将 `application` 改为 `pageContext`。

■ *PageScope1.jsp*

```
<%@ page contentType="text/html; charset=GB2312" %>

<html>
<head>
  <title>CH5 - PageScope1.jsp</title>
</head>
<body>

  <h2>Page 范围 - pageContext</h2>
<%
```

JSP2.0 技术手册

```
        pageContext.setAttribute("Name", "mike");
        pageContext.setAttribute("Password", "browser");
    %>
<jsp:forward page="PageScope2.jsp" />

</body>
</html>
```

■ PageScope2.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>

<html>
<head>
    <title>CH5 - PageScope2.jsp</title>
</head>
<body>

<h2>Page 范围 - pageContext</h2>
<br>
<%
    String Name = (String)pageContext.getAttribute("Name");
    String Password = (String)pageContext.getAttribute("Password");
    out.println("Name = " + Name);
    out.println("Password = " + Password);
%>

</body>
</html>
```

执行结果如图 5-2 所示。



图 5-2 PageScope1.jsp 的执行结果

这个范例程序和之前有点类似，只是之前的程序是 application，现在改为 pageContext，但是结果却大不相同，PageScope2.jsp 根本无法取得 PageScope1.jsp 设定的 Name 和 Password 值，因为在 PageScope1.jsp 当中，是把 Name 和 Password 的属性范围设为 Page，所以 Name 和 Password 的值只能在 PageScope1.jsp 当中取得。若修改 PageScope1.jsp 的程序，重新命名为 PageScope3.jsp，如下：

■ PageScope3.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>

<html>
<head>
  <title>CH5 - PageScope3.jsp</title>
</head>
<body>

<h2>Page 范围 - pageContext</h2>
</br>
<%
    pageContext.setAttribute("Name", "mike");
    pageContext.setAttribute("Password", "browser");

    String Name = (String)pageContext.getAttribute("Name");
    String Password = (String)pageContext.getAttribute("Password");

    out.println("Name = " + Name);
    out.println("Password = " + Password);
%>

</body>
</html>
```

PageScope3.jsp 的执行结果如图 5-3 所示。



图 5-3 PageScope3.jsp 的执行结果

经过修改后的程序，Name 和 Password 的值就能顺利显示出来。这个范例主要用来说明一个概念：若数据设为 Page 范围时，数据只能在同一个 JSP 网页上取得，其他 JSP 网页却无法取得该数据。

5-1-2 JSP Scope—Request

接下来介绍第二种范围：Request。Request 的范围是指在一 JSP 网页发出请求到另一个 JSP 网页之间，随后这个属性就失效。设定 Request 的范围时可利用 request 对象中的 setAttribute()

和 `getAttribute()`。我们再来看下列这个范例：

■ *RequestScope1.jsp*

```
<%@ page contentType="text/html; charset=GB2312" %>

<html>
<head>
  <title>CH5 - RequestScope1.jsp</title>
</head>
<body>

<h2>Request 范围 - request</h2>

<%
    request.setAttribute("Name", "mike");
    request.setAttribute("Password", "browser");
%>
<jsp:forward page="RequestScope2.jsp"/>

</body>
</html>
```

■ *RequestScope2.jsp*

```
<%@ page contentType="text/html; charset=GB2312" %>

<html>
<head>
  <title>CH5 - RequestScope2.jsp</title>
</head>
<body>

<h2>Request 范围 - request</h2>

<%
    String Name = (String) request.getAttribute("Name");
    String Password = (String) request.getAttribute("Password");
    out.println("Name = " + Name);
    out.println("Password = " + Password);
%>

</body>
</html>
```

RequestScope1.jsp 的执行结果如图 5-4 所示。

现在将 `Name` 和 `Password` 的属性范围设为 `Request`，当 *RequestScope1.jsp* 转向到 *RequestScope2.jsp* 时，*RequestScope2.jsp* 也能取得 *RequestScope1.jsp* 设定的 `Name` 和 `Password` 值。不过其他的 JSP 网页无法得到 `Name` 和 `Password` 值，除非它们也和 *RequestScope1.jsp* 有请求的关系。



图 5-4 RequestScope1.jsp 的执行结果

除了利用转向(forward)的方法可以存取 request 对象的数据之外,还能使用包含(include)的方法。

假若我将 *RequestScope1.jsp* 的

```
<jsp:forward page="RequestScope2.jsp" />
```

改为

```
<jsp:include page="RequestScope2.jsp" flush="true" />
```

执行 *RequestScope1.jsp* 时,结果还是和图 5-4 一样。表示使用<jsp:include>标签所包含进来的网页,同样也可以取得 Request 范围的数据。

5-1-3 JSP Scope—Session、Application

表 5-2 介绍了最后两种范围: Session、Application。

表 5-2

范 围	说 明
Session	Session 的作用范围为一段用户持续和服务器所连接的时间,但与服务器断线后,这个属性就无效。只要将数据存入 session 对象,数据的范围就为 Session
Application	Application 的作用范围在服务器一开始执行服务,到服务器关闭为止。Application 的范围最大、停留的时间也最久,所以使用时要特别注意,不然可能会造成服务器负载越来越重的情况。只要将数据存入 application 对象,数据的 Scope 就为 Application

表 5-3 列出了一般储存和取得属性的方法,以下 pageContext、request、session 和 application 皆可使用。

注意
pageContext 并无 getAttributeNames()方法。

表 5-3

方 法	说 明
<code>void setAttribute(String name, Object value)</code>	设定 name 属性的值为 value
<code>Enumeration getAttributeNamesInScope(int scope)</code>	取得所有 scope 范围的属性
<code>Object getAttribute(String name)</code>	取得 name 属性的值
<code>void removeAttribute(String name)</code>	移除 name 属性的值

当我们使用 `getAttribute(String name)` 取得 name 属性的值时，它会回传一个 `java.lang.Object`，因此，我们还必须根据 name 属性值的类型做转换类型(Casting)的工作。例如：若要取得 String 类型的 Name 属性时：

```
String Name = (String)pageContext.getAttribute("Name");
```

若是 Integer 类型的 Year 属性时：

```
Integer Year = (Integer)session.getAttribute("Year");
```

到目前已大约介绍完 JSP 中四种范围(Scope)：Page、Request、Session 和 Application。假若我的数据要设为 Page 范围时，则只需要：

```
pageContext.setAttribute("Year", new Integer(2001));
```

若要为 Request、Session 或 Application 时，就分别存入 request、session 或 application 对象之中，如下：

```
request.setAttribute("Month", new Integer(12) );  
session.setAttribute("Day", new Integer(27) );  
application.setAttribute("Times", new Integer(10));
```

接下来就正式进入本章的主题：隐含对象(Implicit Object)。

5-2 与 Servlet 有关的隐含对象

与 Servlet 有关的隐含对象有两个：page 和 config。page 对象表示 Servlet 本身；config 对象则是存放 Servlet 的初始参数值。

■ page 对象

page 对象代表 JSP 本身，更准确地说，它代表 JSP 被转译后的 Servlet，因此，它可以调用 Servlet 类所定义的方法，不过实际上，page 对象很少在 JSP 中使用。我们来看看以下的范例程序：

■ PageInfo.jsp

```
<%@ page info="JSP 2.0 技术手册" contentType="text/html; charset=GB2312" %>
```

JSP2.0 技术手册


```

<html>
<head>
  <title>CH5 - PageInfo.jsp</title>
</head>
<body>

<h2>page 隐含对象</h2>
Page Info = <%= ((javax.servlet.jsp.HttpJspPage)page).getServletInfo() %>

</body>
</html>

```

这个例子中，我们先设定 page 指令的 info 属性为“JSP 2.0 技术手册”，page 对象的类型为 java.lang.Object，我们调用 javax.servlet.jsp.HttpJspPage 中 getServletInfo() 的方法，将 Info 打印出来，执行结果如图 5-5 所示。



图 5-5 PathInfo.jsp 的执行结果

■ config 对象

config 对象里存放着一些 Servlet 初始的数据结构，config 对象和 page 对象一样都很少被用到。config 对象实现于 javax.servlet.ServletConfig 接口，它共有下列四种方法：

```

public String getInitParameter(name)
public java.util.Enumeration getInitParameterNames( )
public ServletContext getServletContext( )
public String getServletName( )

```

上述前两种方法可以让 config 对象取得 Servlet 初始参数值，如果此数值不存在，就传回 null。例如：当我们在 web.xml 中设定如下时：

```

<?xml version="1.0" encoding="ISO-8859-1"?>

<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
  version="2.4">

```

JSP2.0 技术手册

```
:
<servlet>
  <servlet-name>ServletConfigurator</servlet-name>
  <servlet-class>
    org.logicalcobwebs.proxool.configuration.ServletConfigurator
  </servlet-class>

  <init-param>
    <param-name>propertyFile</param-name>
    <param-value>
      WEB-INF/classes/Proxool.properties
    </param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
:
</web-app>
```

那么我们就可以直接使用 `config.getInitParameter("propertyFile")` 来取得名称为 `propertyFile`、其值为 `WEB-INF/classes/Proxool.properties` 的参数。如下范例：

```
String propertyFile = (String)config.getInitParameter("propertyFile");
```

5-3 与 Input / Output 有关的隐含对象

本节中，我们将介绍和 Input/Output 有关的隐含对象，它们包括：`out`、`request` 和 `response` 对象。`request` 对象表示客户端请求的内容；`response` 对象表示响应客户端的结果；而 `out` 对象负责把数据的结果显示到客户端的浏览器。

■ request 对象

`request` 对象包含所有请求的信息，如：请求的来源、标头、cookies 和请求相关的参数值等等。在 JSP 网页中，`request` 对象是实现 `javax.servlet.http.HttpServletRequest` 接口的，`HttpServletRequest` 接口所提供的方法，可以将它分为四大类：

- (1) 在 5-1-3 小节提到的储存和取得属性方法；
- (2) 能够取得请求参数的方法，如表 5-4：

表 5-4 取得请求参数的方法

方 法	说 明
<code>String getParameter(String name)</code>	取得 <code>name</code> 的参数值
<code>Enumeration getParameterNames()</code>	取得所有的参数名称
<code>String [] getParameterValues(String name)</code>	取得所有 <code>name</code> 的参数值
<code>Map getParameterMap()</code>	取得一个要求参数的 Map

JSP2.0 技术手册

(3) 能够取得请求 HTTP 标头的方法，如表 5-5：

表 5-5 取得请求标头的方法

方 法	说 明
String getHeader(String name)	取得 name 的标头
Enumeration getHeaderNames()	取得所有的标头名称
Enumeration getHeaders(String name)	取得所有 name 的标头
int getIntHeader(String name)	取得整数类型 name 的标头
long getDateHeader(String name)	取得日期类型 name 的标头
Cookie[] getCookies()	取得与请求有关的 cookies

(4) 其他的方法，例如：取得请求的 URL、IP 和 session，如表 5-6：

表 5-6 其他请求的方法

方 法	说 明
String getContextPath()	取得 Context 路径(即站台名称)
String getMethod()	取得 HTTP 的方法(GET、POST)
String getProtocol()	取得使用的协议 (HTTP/1.1、HTTP/1.0)
String getQueryString()	取得请求的参数字符串，不过，HTTP 的方法必须为 GET
String getRequestedSessionId()	取得用户端的 Session ID
String getRequestURI()	取得请求的 URL，但是不包括请求的参数字符串
String getRemoteAddr()	取得用户的 IP 地址
String getRemoteHost()	取得用户的主机名称
int getRemotePort()	取得用户的主机端口
String getRemoteUser()	取得用户的名称
void setCharacterEncoding(String encoding)	设定编码格式，用来解决窗体传递中文的问题

我们来看下面这个程序范例，相信对读者会更加有帮助。

■ Request.html

```
<html>
<head>
  <title>CH5 - Request.html</title>
<meta http-equiv="Content-Type" content="text/html; charset=GB2312">
</head>
<body>

  <form action="Request.jsp" method="GET">
Name: <input type="text" name="Name" size="20" maxlength="20"><br>
Number: <input type="text" name="Number" size="20" maxlength="20"><br><br>
<input type="submit" value="传送">
```

JSP2.0 技术手册

```
</form>
```

```
</body>
```

```
</html>
```

Request.html 的执行结果如图 5-6 所示, 笔者在 Name 的字段中输入 browser; Number 字段中输入 123456789。



图 5-6 Request.html 的执行结果

■ Request.jsp

```
<%@ page language="java" contentType="text/html; charset=GB2312" %>
<html>
<head>
  <title>CH5 - Request.jsp</title>
</head>
<body>

<h2>javax.servlet.http.HttpServletRequest 接口所提供的方法</h2>

getParameter("Name"): <%= request.getParameter("Name") %><br>
getParameter("Number"): <%= request.getParameter("Number") %><br>
getAttribute("Name"): <%= request.getAttribute("Name") %><br>
getAttribute("Number"): <%= request.getAttribute("Number") %><br><br>

getAuthType(): <%= request.getAuthType() %><br>
getProtocol(): <%= request.getProtocol() %><br>
getMethod(): <%= request.getMethod() %><br>
getScheme(): <%= request.getScheme() %><br>
getContentType(): <%= request.getContentType() %><br>
getContentLength(): <%= request.getContentLength() %><br>
getCharacterEncoding(): <%= request.getCharacterEncoding() %><br>
getRequestedSessionId(): <%= request.getRequestedSessionId() %><br><br>
```

```
getContextPath(): <%= request.getContextPath() %><br>
getServletPath(): <%= request.getServletPath() %><br>
getPathInfo(): <%= request.getPathInfo() %><br>
getRequestURI(): <%= request.getRequestURI() %><br>
getQueryString(): <%= request.getQueryString() %><br><br>

getRemoteAddr(): <%= request.getRemoteAddr() %><br>
getRemoteHost(): <%= request.getRemoteHost() %><br>
getRemoteUser(): <%= request.getRemoteUser() %><br>
getRemotePort(): <%= request.getRemotePort() %><br>
getServerName(): <%= request.getServerName() %><br>
getServerPort(): <%= request.getServerPort() %><br>

</body>
</html>
```

Request.jsp 的执行结果如图 5-7 所示。

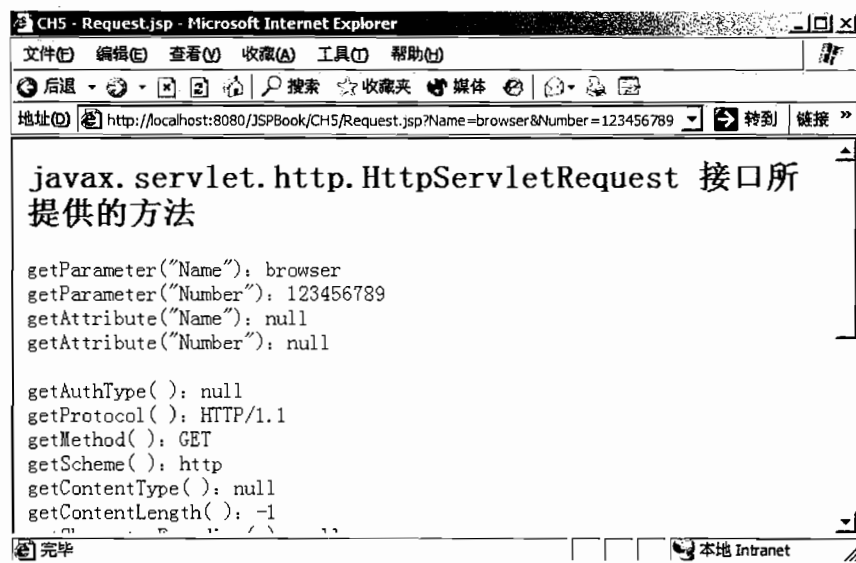


图 5-7 Request.jsp 的执行结果

在 *Request.jsp* 中, 使用 `request.getParameter("Name")` 和 `request.getParameter("Number")`, 能够取得 *Request.html* 窗体的值。除了取得请求的参数值之外, 也能取得一些相关信息, 如: 使用的协议、方法、URI 等等。

■ response 对象

`response` 对象主要将 JSP 处理数据后的结果传回到客户端。`response` 对象是实现

JSP2.0 技术手册

javax.servlet.http.HttpServletResponse 接口。表 5-7、表 5-8、表 5-9 列出了 response 对象的方法。

表 5-7 设定表头的方法

方 法	说 明
void addCookie(Cookie cookie)	新增 cookie
void addDateHeader(String name, long date)	新增 long 类型的值到 name 标头
void addHeader(String name, String value)	新增 String 类型的值到 name 标头
void addIntHeader(String name, int value)	新增 int 类型的值到 name 标头
void setDateHeader(String name, long date)	指定 long 类型的值到 name 标头
void setHeader(String name, String value)	指定 String 类型的值到 name 标头
void setIntHeader(String name, int value)	指定 int 类型的值到 name 标头

表 5-8 设定响应状态码的方法

方 法	说 明
void sendError(int sc)	传送状态码(status code)
void sendError(int sc, String msg)	传送状态码和错误信息
void setStatus(int sc)	设定状态码

表 5-9 用来 URL 重写(rewriting)的方法

方 法	说 明
String encodeRedirectURL(String url)	对使用 sendRedirect()方法的 URL 予以编码

有时候,当我们修改程序后,产生的结果却是之前的数据,执行浏览器上的刷新,才能看到更改数据后的结果,针对这个问题,有时是因为浏览器会将之前浏览过的数据存放在浏览器的 cache 中,所以当我们再次执行时,浏览器会直接从 cache 中取出,因此,会显示之前旧的数据。笔者将写一个 *Non-cache.jsp* 程序来解决这个问题。

■ Non-cache.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>

<html>
<head>
  <title>CH5 - Non-cache.jsp</title>
</head>
<body>

<h2>解决浏览器 cache 的问题 - response</h2>
<%
  if (request.getProtocol().compareTo("HTTP/1.0") == 0)
    response.setHeader("Pragma", "no-cache");
  else if (request.getProtocol().compareTo("HTTP/1.1") == 0)
```

JSP2.0 技术手册


```

        response.setHeader("Cache-Control", "no-cache");

        response.setDateHeader("Expires", 0);
    %>

</body>
</html>

```

先用 request 对象取得协议, 如果为 HTTP/1.0, 就设定标头, 内容为 setHeader("Pragma", "no-cache"); 若为 HTTP/1.1, 就设定标头为 response.setHeader("Cache-Control", "no-cache"), 最后再设定 response.setDateHeader("Expires", 0)。这样 *Non-cache.jsp* 网页在浏览过后, 就不会再存放到浏览器或是 proxy 服务器的 cache 中。表 5-10 列出了 HTTP/1.1 Cache-Control 标头的设定参数:

表 5-10 HTTP/1.1 Cache-Control 标头的设定参数

参 数	说 明
public	数据内容皆被储存起来, 就连有密码保护的网页也是一样, 因此安全性相当低
private	数据内容只能被储存到私有的 caches, 即 non-shared caches 中
no-cache	数据内容绝不被储存起来。proxy 服务器和浏览器读到此标头, 就不会将数据内容存入 caches 中
no-store	数据内容除了不能存入 caches 中之外, 亦不能存入暂时的磁盘中, 这个标头防止敏感性的数据被复制
must-revalidate	用户在每次读取数据时, 会再次和原来的服务器确定是否为最新数据, 而不再通过中间的 proxy 服务器
proxy-revalidate	这个参数有点像 must-revalidate, 不过中间接收的 proxy 服务器可以互相分享 caches
max-age=xxx	数据内容在经过 xxx 秒后, 就会失效, 这个标头就像 Expires 标头的功能一样, 不过 max-age=xxx 只能服务 HTTP/1.1 的用户。假设两者并用时, max-age=xxx 有较高的优先权

有时候, 我们想要让网页自己能自动更新, 因此, 须使用到 Refresh 这个标头。举个例子, 我们告诉浏览器, 每隔三分钟, 就重新加载此网页:

```
response.setIntHeader("Refresh", 180)
```

如果想要过十秒后, 调用浏览器转到 <http://Server/Path> 的网页时, 可用如下代码:

```
response.setHeader("Refresh", "10; URL=http://Server/Path" )
```

如果大家对 HTML 语法还熟悉, 则 HTML 语法中也有类似的功能:

```
<META HTTP-EQUIV="Refresh" CONTENT=" 10; URL=http://Server/Path" >
```

上述两种方法皆可以做到自动重新加载。

JSP2.0 技术手册

■ out 对象

out 对象能把结果输出到网页上。通常我们最常使用 `out.println(String name)` 和 `out.print(String name)`，它们两者最大的差别在于 `println()` 在输出的数据后面会自动加上换行的符号，例如：你在 Dos Console 的窗口下，发现到它输出数据后会自动换行；反之，`print()` 不会在数据后自动换行。

out 对象除了这两种方法最常使用之外，它还有一些方法（见表 5-11），这些方法主要是用来控制管理输出的缓冲区(buffer)和输出流(output stream)。

表 5-11 out 对象方法

方 法	说 明
<code>void clear()</code>	清除输出缓冲区的内容
<code>void clearBuffer()</code>	清除输出缓冲区的内容
<code>void close()</code>	关闭输出流，清除所有的内容
<code>int getBufferSize()</code>	取得目前缓冲区的大小(KB)
<code>int getRemaining()</code>	取得目前使用后还剩下的缓冲区大小(KB)
<code>boolean isAutoFlush()</code>	如果回传为 true，表示如缓冲区满了，会自动清除；若为 false，表示如果缓冲区满了，不会自动清除，而会产生异常处理

我们在这里举个例子，说明如何知道目前输出缓冲区的大小。

■ Out.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>

<html>
<head>
  <title>CH5 - Out.jsp</title>
</head>
<body>

<h2>javax.servlet.jsp.JspWriter - out 对象</h2>
<%
    int BufferSize = out.getBufferSize();
    int Available = out.getRemaining();
    int Used = BufferSize - Available;
%>
BufferSize : <%= BufferSize %><br>
Available : <%= Available %><br>
Used : <%= Used %><br>

</body>
</html>
```

BufferSize 是一开始默认缓冲区的大小，默认值为 8KB；Available 则是表示经过程的

执行, 目前缓冲区还剩下多少可以使用; 而 Used 则表示我们使用了多少的缓冲区。Out.jsp 执行结果如图 5-8。

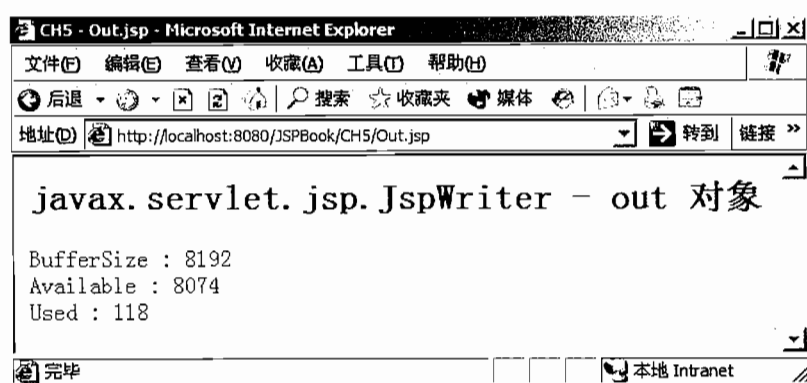


图 5-8 Out.jsp 的执行结果

5-4 与 Context 有关的隐含对象

在本节中, 我们要介绍 session、application、pageContext 这三个对象。session 对象提供一些机制, 让服务器能个别辨认用户。当程序在执行时, application 对象能提供服务端 (Server-Side) 的 Context, 说明哪些资源是可利用的, 哪些信息是可获取的。pageContext 对象提供存取所有在此网页中可被利用的隐含对象, 并且可以管理它们的属性。

■ session 对象

session 对象表示目前个别用户的会话(session)状况, 用此项机制可以轻易识别每一个用户, 然后针对每一个别用户的要求, 给予正确的响应。例如: 购物车最常使用 session 的概念, 当用户把物品放入购物车时, 他不须重复做身份确认的动作(如: Login), 就能把物品放入用户的购物车。但服务器利用 session 对象, 就能确认用户是谁, 把它的物品放在属于用户的购物车, 而不会将物品放错到别人的购物车。除了购物车之外, session 对象也通常用来做追踪用户的功能, 这在第十章有更加详细的说明。

session 对象实现 javax.servlet.http.HttpSession 接口, 表 5-12 列出了一些常用的方法。

表 5-12 javax.servlet.http.HttpSession 接口所提供的方法

方 法	说 明
long getCreationTime()	取得 session 产生的时间, 单位是毫秒, 由 1970 年 1 月 1 日零时算起
String getId()	取得 session 的 ID

续表

方 法	说 明
long getLastAccessedTime()	取得用户最后通过这个 session 送出请求的时间, 单位是毫秒, 由 1970 年 1 月 1 日零时算起
long getMaxInactiveInterval()	取得最大 session 不活动的时间, 若超过这时间, session 将会失效, 时间单位为秒
void invalidate()	取消 session 对象, 并将对象存放的内容完全抛弃
boolean isNew()	判断 session 是否为"新"的, 所谓"新"的 session, 表示 session 已由服务器产生, 但是 client 尚未使用
void setMaxInactiveInterval(int interval)	设定最大 session 不活动的时间, 若超过这时间, session 将会失效, 时间单位为秒

session 对象也可以储存或取得用户相关的数据, 例如: 用户的名称、用户所订购的物品、用户的权限, 等等, 这些要看我们的程序如何去设计。例如: 我要设定某些网页必须要求用户先做登录(Login)的动作, 确定是合法的用户时, 才允许读取网页内容, 否则把网页重新转向到登录的网页上。

■ Login.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>

<html>
<head>
  <title>CH5 - Login.jsp</title>
</head>
<body>

  <h2>javax.servlet.http.HttpSession - session 对象</h2>
  <form action=Login.jsp method="POST" >
    Login Name: <input type="text" name="Name"><br>
    Login Password: <input type="text" name="Password" ><br>
    <input type="submit" value="Send"><br>
  </form>

  <% if (request.getParameter("Name") != null &&
    request.getParameter("Password") != null) {
    String Name = request.getParameter("Name");
    String Password = request.getParameter("Password");

    if (Name.equals("mike") && Password.equals("1234")) {
      session.setAttribute("Login", "OK");
      response.sendRedirect("Member.jsp");
    }
    else {
      out.println("登录错误, 请输入正确名称");
    }
  }
  %>
```

JSP2.0 技术手册

```
</body>
</html>
```

在 *Login.jsp* 的程序中,我要求用户分别输入名称和密码,如果输入的名称和密码分别为 mike 和 1234 时,就把名称为 Login、其值为 OK 的属性,加入到 session 对象当中,然后进入 *Member.jsp* 网页,如图 5-9;若输入错误时,就显示出“登录错误,请输入正确名称”。不允许登录至 *Member.jsp*,如图 5-10 所示。



图 5-9 登录成功,顺利进入 Member.jsp

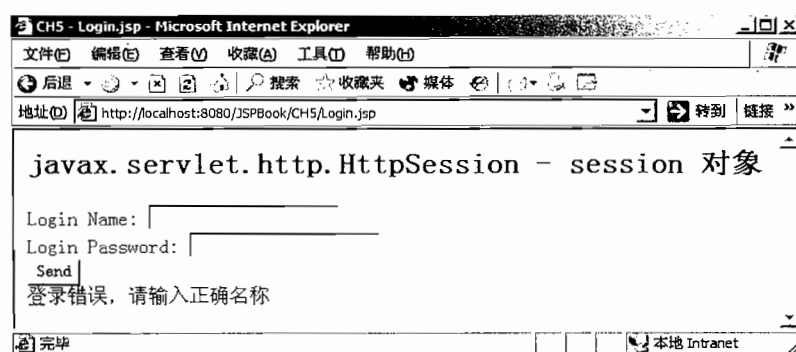


图 5-10 登录失败画面

这时大家一定会想,如果我不通过 *Login.jsp* 网页,直接执行 *Member.jsp*,那不就能够进去了。没错,因此我们还要在 *Member.jsp* 中加入一段程序代码,来确认用户是否有先通过 *Login.jsp* 的身份确认,然后再到 *Member.jsp* 中。*Member.jsp* 程序如下:

■ Member.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>

<html>
<head>
  <title>CH5 - Member.jsp</title>
</head>
<body>
```

JSP2.0 技术手册

```

<h2>javax.servlet.http.HttpSession - session 对象</h2>
<%
    String Login = (String)session.getAttribute("Login");

    if (Login != null && Login.equals("OK")) {
        out.println("欢迎进入");
        session.invalidate();
    }
    else {
        out.println("请先登录, 谢谢");
        out.println("<br>经过五秒之后, 网页会自动返回 Login.jsp");

        response.setHeader("Refresh", "5;URL=Login.jsp");
    }
%>
</body>
</html>

```

在 *Member.jsp* 中我利用 `session.getAttribute("Login")`, 如果用户是通过 *Login.jsp* 网页进入, 并且顺利通过身份确认取得 `Login=OK`, 到 *Member.jsp* 再做确认时, 也能顺利通过; 否则, 如果直接连接到 *Member.jsp* 时, `Login` 的值会等于 `NULL`, 则程序经过五秒后, 重新加载 *Login.jsp*, 要求用户先行登录。若直接执行 *Member.jsp*, 而没有经过登录手续时, 就会发现如图 5-11。



图 5-11 直接执行 Member.jsp, 并未经过登录手续

最后要提醒读者一点, `session` 对象不像其他的隐含对象, 可以在任何的 JSP 网页中使用, 如果在 JSP 网页中, `page` 指令的属性 `session` 设为 `false` 时, 使用 `session` 对象就会产生编译错误 (`javax.servlet.ServletException: Compilation error occurred`), 如下所示:

```

<%@ page session="false" %>
<%
    String Login = (String)session.getAttribute("Login");
    ...
    ...
%>

```

在本书“第十章: Session Tracking”中对 `session` 有更多更详细的介绍。

■ application 对象

application 对象实现 javax.servlet.ServletContext 接口,它主要功用在于取得或更改 Servlet 的设定。下面程序用来说明 JSP 网页被编译成 Servlet 时, application 对象是如何初始化的:

```
pageContext = JspFactory.getPageContext ( this , request , response ,
                                           "errorpage.jsp" , true , 8192 , true );

application = pageContext.getServletContext ( );
```

你可以看到产生的 Servlet 取得了目前的 ServletContext, 并且将它储存在 application 对象当中。application 对象拥有 Application 的范围, 意思就是说它的生命周期是由服务器产生开始至服务器关机为止。表 5-13、表 5-14、表 5-15 列出了其相关方法:

表 5-13 javax.servlet.ServletContext 接口容器相关信息的方法

方 法	说 明
int getMajorVersion()	取得 Container 主要的 Servlet API 版本, 如: 2
int getMinorVersion()	取得 Container 次要的 Servlet API 版本, 如: 4
String getServerInfo()	取得 Container 的名称和版本

```
<%= application.getMajorVersion() %><br>
<%= application.getMinorVersion() %><br>
<%= application.getServerInfo() %><br>
```

上述的 getMajorVersion() 和 getMinorVersion() 是取得 Servlet Engine 的版本信息, 假如想要取得 JSP 容器的版本信息, 则可能就要使用到下面这段程序代码:

■ GetJspVersion.jsp

```
<%@ page import="javax.servlet.jsp.JspFactory"
contentType="text/html; charset=GB2312" %>
<html>
<head>
<title>CH5 - GetJspVersion.jsp</title>
</head>
<body>
<h2>取得 JSP Container 版本 - JspFactory 对象</h2>
<%
    JspFactory factory = JspFactory.getDefaultFactory();
    out.println("JSP v 2.0"+
               factory.getEngineInfo().getSpecificationVersion());
%>
</body>
</html>
```

JSP2.0 技术手册

执行结果如图 5-12 所示。

表 5-14 javax.servlet.ServletContext 接口有关服务端的路径和文件的方法

方 法	说 明
String getMimeType(String file)	取得指定文件的 MIME 类型
ServletContext getContext(String uripath)	取得指定 Local URL 的 Application context
String getRealPath(String path)	取得本地端 path 的绝对路径

范例：

```
<%= application.getMimeType("MyFile") %>
<%= application.getContext("/") %>
<%= application.getRealPath("/") %>
```

表 5-15 javax.servlet.ServletContext 接口有关信息记录的方法

方 法	说 明
void log(String message)	将信息写入 log 文件中
void log(String message, Throwable throwable)	将 stack trace 所产生的异常信息写入 log 文件中

application 对象最常被使用在存取环境的信息,因为环境的信息通常都储存在 ServletContext 中,所以常利用 application 对象来存取 ServletContext 中的信息。

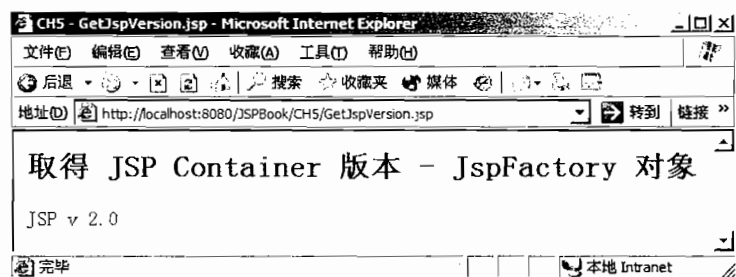


图 5-12 GetJspVersion.jsp 的执行结果

■ pageContext 对象

pageContext 对象能够存取其他隐含对象。当隐含对象本身也支持属性时,pageContext 对象也提供存取那些属性的方法。不过在使用下列方法时,需要指定范围的参数:

```
Object getAttribute(String name, int scope)
Enumeration getAttributeNamesInScope(int scope)
void removeAttribute(String name, int scope)
void setAttribute(String name, Object value, int scope)
```

JSP2.0 技术手册

范围参数有四个常数, 分别代表四种范围: `PAGE_SCOPE` 代表 Page 范围, `REQUEST_SCOPE` 代表 Request 范围, `SESSION_SCOPE` 代表 Session 范围, 最后 `APPLICATION_SCOPE` 代表 Application 范围 (见表 5-16、表 5-17、表 5-18)。

表 5-16 javax.servlet.jsp.PageContext 类取得其他隐含对象的方法

方 法	说 明
Exception getException()	回传目前网页的异常, 不过此网页要为 error page, 例如: exception 隐含对象
JspWriter getOut()	回传目前网页的输出流, 例如: out 隐含对象
Object getPage()	回传目前网页的 Servlet 实体(instance), 例如: page 隐含对象
ServletRequest getRequest()	回传目前网页的请求, 例如: request 隐含对象
ServletResponse getResponse()	回传目前网页的响应, 例如: response 隐含对象
ServletConfig getServletConfig()	回传目前此网页的 ServletConfig 对象, 例如: config 隐含对象
ServletContext getServletContext()	回传目前此网页的执行环境(context), 例如: application 隐含对象
HttpSession getSession()	回传和目前网页有联系的会话(session), 例如: session 隐含对象

表 5-17 javax.servlet.jsp.PageContext 类所提供取得属性的方法

方 法	说 明
Object getAttribute(String name, int scope)	回传 name 属性, 范围为 scope 的属性对象, 回传类型为 java.lang.Object
Enumeration getAttributeNamesInScope(int scope)	回传所有属性范围为 scope 的属性名称, 回传类型为 Enumeration
int getAttributesScope(String name)	回传属性名称为 name 的属性范围
void removeAttribute(String name)	移除属性名称为 name 的属性对象
void removeAttribute(String name, int scope)	移除属性名称为 name, 范围为 scope 的属性对象
void setAttribute(String name, Object value, int scope)	指定属性对象的名称为 name、值为 value、范围为 scope
Object findAttribute(String name)	寻找在所有范围中属性名称为 name 的属性对象

表 5-18 javax.servlet.jsp.PageContext 类所提供范围的变量

常 数	说 明
PAGE_SCOPE	存入 pageContext 对象的属性范围
REQUEST_SCOPE	存入 request 对象的属性范围
SESSION_SCOPE	存入 session 对象的属性范围
APPLICATION_SCOPE	存入 application 对象的属性范围

接下来示范一个小程序，让读者能够更加明白。

■ PageContext.jsp

```
<%@ page import="java.util.Enumeration"
contentType="text/html;charset=GB2312" %>

<html>
<head>
  <title>CH5 - PageContext.jsp</title>
</head>
<body>

<h2>javax.servlet.jsp.PageContext - pageContext </h2>

<%
  Enumeration enum =
    pageContext.getAttributeNamesInScope(PageContext.APPLICATION_SCOPE );

  while (enum.hasMoreElements())
  {
    out.println("application attribute: "+enum.nextElement()+"<br>");
  }
%>

</body>
</html>
```

PageContext.jsp 主要目的是：在这页当中，取得所有属性范围为 Application 的属性名称，然后再依序显示出来这些属性。

首先要记得导入 `java.util.Enumeration`。`pageContext.getAttributeNamesInScope()` 会回传所有指定范围的属性名称，因此，我们产生 `Enumeration` 对象 `enum`，利用 `enum` 来收集所有属性范围为 Application 的数据，然后再一一地取出打印出来。这里最重要的是让读者了解如何设定 `scope` 的参数，因此下面这行代码：

```
PageContext.APPLICATION_SCOPE
```

是最主要的。有了这个范例程序之后，读者应该能够快速学会使用 `pageContext` 对象所提供的方法。

`pageContext` 对象除了提供上述的方法之外，另外还有两种方法：`forward (String Path)`、`include (String Path)`，这两种方法的功能和之前提到的 `<jsp:forward>` 与 `<jsp:include>` 相似，因此在这也不多加讨论。

5-5 与 Error 有关的隐含对象

最后一类的隐含对象只有一个成员：`exception` 对象。当 JSP 网页有错误时会产生异常，而

exception 对象就来针对这个异常做处理。

■ exception 对象

exception 对象和 session 对象一样，并不是在每一个 JSP 网页中都能够使用。若要使用 exception 对象时，必须在 page 指令中设定。

```
<%@ page isErrorPage="true" %>
```

才能使用，不然在编译时会产生错误。

■ Exception.jsp

```
<%@ page contentType="text/html; charset=GB2312" isErrorPage="true" %>

<html>
<head>
  <title>CH5 - Exception.jsp</title>
</head>
<body>

<h2> exception 对象</h2>

Exception: <%= exception %><br>
Message: <%= exception.getMessage() %><br>
Localized Message: <%= exception.getLocalizedMessage() %><br>
Stack Trace: <% exception.printStackTrace(new java.io.PrintWriter(out)); %><br>

</body>
</html>
```

一般 error page 的程序代码和 *Exception.jsp* 程序相似，它已经将所有该打印出来的错误信息包括进来。在这段程序代码中使用了三个方法：getMessage()、getLocalizedMessage()、printStackTrace(new java.io.PrintWriter(out))，其中 printStackTrace() 的参数要为 PrintWriter 而不是 JspWriter。

6

第六章

Expression Language

本章将分以下 8 节，详细介绍 Expression Language 的语法和使用：

- 6-1 EL 简介
- 6-2 EL 语法
- 6-3 EL 隐含对象
- 6-4 EL 算术运算符
- 6-5 EL 关系运算符
- 6-6 EL 逻辑运算符
- 6-7 EL 其他运算符
- 6-8 EL Functions

JSP2.0 技术手册

6-1 EL 简介

EL 全名为 Expression Language，它原本是 JSTL 1.0 为方便存取数据所自定义的语言。当时 EL 只能在 JSTL 标签中使用，如下：

```
<c:out value="${ 3 + 7}">
```

程序执行结果为 10。但是你却不能直接在 JSP 网页中使用：

```
<p>Hi ! ${ username }</p>
```

到了 JSP 2.0 之后，EL 已经正式纳入成为标准规范之一。因此，只要是支持 Servlet 2.4 / JSP 2.0 的 Container，就都可以在 JSP 网页中直接使用 EL 了。

除了 JSP 2.0 建议使用 EL 之外，JavaServer Faces(JSR-127) 也考虑将 EL 纳入规范，由此可知，EL 如今已经是一项成熟、标准的技术。

注意

假若您所用的 Container 只支持 Servlet 2.3/JSP 1.2，如：Tomcat 4.1.29，您就不能在 JSP 网页中直接使用 EL，必须安装支持 Servlet 2.4 / JSP 2.0 的 Container。

6-2 EL 语法

EL 语法很简单，它最大的特点就是使用上很方便。接下来介绍 EL 主要的语法结构：

```
${sessionScope.user.sex}
```

所有 EL 都是以 \${ 为起始、以 } 为结尾的。上述 EL 范例的意思是：从 Session 的范围中，取得用户的性别。假若依照之前 JSP Scriptlet 的写法如下：

```
User user = (User)session.getAttribute("user");  
String sex = user.getSex( );
```

两者相比较之下，可以发现 EL 的语法比传统 JSP Scriptlet 更为方便、简洁。

6-2-1 . 与 [] 运算符

EL 提供 . 和 [] 两种运算符来存取数据。下列两者所代表的意思是相同的：

```
${sessionScope.user.sex}
```

等于

```
${sessionScope.user["sex"]}
```

. 和 [] 也可以同时混合使用，如下：

JSP2.0 技术手册

```
${sessionScope.shoppingCart[0].price}
```

回传结果为 shoppingCart 中第一项物品的价格。

不过，以下两种情况，两者会有差异：

(1) 当要存取的属性名称中包含一些特殊字符，如 `.` 或 `-` 等非字母或数字的符号，就一定要使用 `[]`，例如：

```
${user.My-Name }
```

上述是不正确的方式，应当改为：

```
${user["My-Name"] }
```

(2) 我们来考虑下列情况：

```
${sessionScope.user[data]}
```

此时，`data` 是一个变量，假若 `data` 的值为 `"sex"` 时，那上述的例子等于 `${sessionScope.user.sex}`；假若 `data` 的值为 `"name"` 时，它就等于 `${sessionScope.user.name}`。因此，如果要动态取值时，就可以用上述的方法来做，但 `.` 无法做到动态取值。

接下来，我们更详细地来讨论一些情况，首先假设有一个 EL：

```
${expr-a[expr-b]}
```

- (1) 当 `expr-a` 的值为 `null` 时，它会回传 `null`。
- (2) 当 `expr-b` 的值为 `null` 时，它会回传 `null`。
- (3) 当 `expr-a` 的值为 `Map` 类型时：
 - 假若 `!value-a.containsKey(value-b)` 为真，则回传 `null`。
 - 否则回传 `value-a.get(value-b)`。
- (4) 当 `expr-a` 的值为 `List` 或 `array` 类型时：
 - 将 `value-b` 的值强制转型为 `int`，假若不能转型为 `int` 时，会产生 `error`。
 - 然后，假若 `value-a.get(value-b)` 或 `Array.get(value-a, value-b)` 产生 `ArrayIndexOutOfBoundsException` 或 `IndexOutOfBoundsException` 时，则回传 `null`。
 - 假若 `value-a.get(value-b)` 或 `Array.get(value-a, value-b)` 产生其他的异常时，则会产生 `error`。
 - 最后都没有任何异常产生时，回传 `value-a.get(value-b)` 或 `Array.get(value-a, value-b)`。
- (5) 当 `expr-a` 的值为 `JavaBean` 对象时：
 - 将 `value-b` 的值强制转型为 `String`。
 - 假若 `getter` 产生异常时，则会产生 `error`。若没有异常产生时，则回传 `getter` 的结果。

6-2-2 EL 变量

EL 存取变量数据的方法很简单，例如：`${username}`。它的意思是取出某一范围中名称为 `username` 的变量。因为我们并没有指定哪一个范围的 `username`，所以它的默认值会先从 `Page` 范围找，假如找不到，再依序到 `Request`、`Session`、`Application` 范围。假如途中找到 `username`，

JSP2.0 技术手册

就直接回传，不再继续找下去，但是假如全部的范围都没有找到时，就回传 `null`（见表 6-1）：

表 6-1

属性范围	在 EL 中的名称
Page	PageScope
Request	RequestScope
Session	SessionScope
Application	ApplicationScope

自动搜索顺序

我们也可以指定要取出哪一个范围的变量（见表 6-2）：

表 6-2

范 例	说 明
<code>\${pageScope.username}</code>	取出 Page 范围的 username 变量
<code>\${requestScope.username}</code>	取出 Request 范围的 username 变量
<code>\${sessionScope.username}</code>	取出 Session 范围的 username 变量
<code>\${applicationScope.username}</code>	取出 Application 范围的 username 变量

其中，`pageScope`、`requestScope`、`sessionScope` 和 `applicationScope` 都是 EL 的隐含对象，由它们的名称可以很容易猜出它们所代表的意思，例如：`${sessionScope.username}` 是取出 Session 范围的 username 变量。这种写法是不是比之前 JSP 的写法：

```
String username = (String) session.getAttribute("username");
```

容易、简洁许多。有关 EL 隐含对象在 6-3 节中有更详细的介绍。

6-2-3 自动转变类型

EL 除了提供方便存取变量的语法之外，它另外一个方便的功能就是：自动转变类型，我们来看下面这个范例：

```
${param.count + 20}
```

假若窗体传来 `count` 的值为 10 时，那么上面的结果为 30。之前没接触过 JSP 的读者可能会认为上面的例子是理所当然的，但是在 JSP 1.2 之中不能这样做，原因是从窗体所传来的值，它们的类型一律是 `String`，所以当你接收之后，必须再将它转为其他类型，如：`int`、`float` 等等，然后才能执行一些数学运算，下面是之前的做法：

```
String str_count = request.getParameter("count");
int count = Integer.parseInt(str_count);
count = count + 20;
```


接下来再详细说明 EL 类型转换的规则：

(1) 将 A 转为 String 类型

- 假若 A 为 String 时：回传 A
- 否则，当 A 为 null 时：回传 ""
- 否则，当 A.toString() 产生异常时：错误！
- 否则，回传 A.toString()

(2) 将 A 转为 Number 类型的 N

- 假若 A 为 null 或 "" 时：回传 0
- 假若 A 为 Character 时：将 A 转为 new Short((short)a.charValue())
- 假若 A 为 Boolean 时：错误！
- 假若 A 为 Number 类型和 N 一样时：回传 A
- 假若 A 为 Number 时：
 - 假若 N 是 BigInteger 时：
 - 假若 A 为 BigDecimal 时：回传 A.toBigInteger()
 - 否则，回传 BigInteger.valueOf(A.longValue())
 - 假若 N 是 BigDecimal 时：
 - 假若 A 为 BigInteger 时：回传 A.toBigDecimal()
 - 否则，回传 BigDecimal.valueOf(A.doubleValue())
 - 假若 N 为 Byte 时：回传 new Byte(A.byteValue())
 - 假若 N 为 Short 时：回传 new Short(A.shortValue())
 - 假若 N 为 Integer 时：回传 new Integer(A.intValue())
 - 假若 N 为 Long 时：回传 new Long(A.longValue())
 - 假若 N 为 Float 时：回传 new Float(A.floatValue())
 - 假若 N 为 Double 时：回传 new Double(A.doubleValue())
 - 否则，错误！
- 假若 A 为 String 时：
 - 假若 N 是 BigDecimal 时：
 - 假若 new BigDecimal(A) 产生异常时：错误！
 - 否则，回传 new BigDecimal(A)
 - 假若 N 是 BigInteger 时：
 - 假若 new BigInteger(A) 产生异常时：错误！
 - 否则，回传 new BigInteger(A)
 - 假若 N.valueOf(A) 产生异常时：错误！
 - 否则，回传 N.valueOf(A)
- 否则，错误！

(3) 将 A 转为 Character 类型

- 假若 A 为 null 或 "" 时：回传 (char)0

- 假若 A 为 Character 时：回传 A
 - 假若 A 为 Boolean 时：错误!
 - 假若 A 为 Number 时：转换为 Short 后，然后回传 Character
 - 假若 A 为 String 时：回传 A.charAt(0)
 - 否则，错误!
- (4) 将 A 转为 Boolean 类型
- 假若 A 为 null 或 "" 时：回传 false
 - 否则，假若 A 为 Boolean 时：回传 A
 - 否则，假若 A 为 String，且 Boolean.valueOf(A)没有产生异常时：回传 Boolean.valueOf(A)
 - 否则，错误!

6-2-4 EL 保留字

EL 的保留字如表 6-3:

表 6-3

And	eq	gt	true
Or	ne	le	false
No	lt	ge	null
instanceof	empty	div	mod

所谓保留字的意思是指变量在命名时，应该避开上述的名字，以免程序编译时发生错误。

6-3 EL 隐含对象

笔者在“第五章：隐含对象（Implicit Object）”中，曾经介绍过 9 个 JSP 隐含对象，而 EL 本身也有自己的隐含对象。EL 隐含对象总共有 11 个（见表 6-4）：

表 6-4

隐含对象	类 型	说 明
PageContext	javax.servlet.ServletContext	表示此 JSP 的 PageContext
PageScope	java.util.Map	取得 Page 范围的属性名称所对应的值
RequestScope	java.util.Map	取得 Request 范围的属性名称所对应的值
sessionScope	java.util.Map	取得 Session 范围的属性名称所对应的值
applicationScope	java.util.Map	取得 Application 范围的属性名称所对应的值
param	java.util.Map	如同 ServletRequest.getParameter(String name)。回传 String 类型的值

JSP2.0 技术手册

续表

隐含对象	类 型	说 明
paramValues	java.util.Map	如同 ServletRequest.getParameterValues(String name)。回传 String []类型的值
header	java.util.Map	如同 ServletRequest.getHeader(String name)。回传 String 类型的值
headerValues	java.util.Map	如同 ServletRequest.getHeaders(String name)。回传 String []类型的值
cookie	java.util.Map	如同 HttpServletRequest.getCookies()
initParam	java.util.Map	如同 ServletContext.getInitParameter(String name)。回传 String 类型的值

这 11 个隐含对象(Implicit Object)，笔者将它分成三类：

- 1. 与范围有关的隐含对象
 - applicationScope
 - sessionScope
 - requestScope
 - pageScope
- 2. 与输入有关的隐含对象
 - param
 - paramValues
- 3. 其他隐含对象
 - cookie
 - header
 - headerValues
 - initParam
 - pageContext

接下来笔者会依照上面的分类顺序，为读者介绍这些隐含对象。

6-3-1 属性(Attribute)与范围(Scope)

与范围有关的 EL 隐含对象包含以下四个：pageScope、requestScope、sessionScope 和 applicationScope，它们基本上就和 JSP 的 pageContext、request、session 和 application 一样，所以笔者在这里只稍略说明。不过必须注意的是，这四个隐含对象只能用来取得范围属性值，即 JSP 中的 getAttribute(String name)，却不能取得其他相关信息，例如：JSP 中的 request 对象除可以存取属性之外，还可以取得用户的请求参数或表头信息等等。但是在 EL 中，它就只能单纯

用来取得对应范围的属性值，例如：我们要在 session 中储存一个属性，它的名称为 username，在 JSP 中使用 `session.getAttribute("username")` 来取得 username 的值，但是在 EL 中，则是使用 `${sessionScope.username}` 来取得其值的。接下来分别对这四个隐含对象做简短的说明：

- pageScope

范围和 JSP 的 Page 相同，也就是单单一页 JSP Page 的范围(Scope)。

- requestScope

范围和 JSP 的 Request 相同，requestScope 的范围是指从一个 JSP 网页请求到另一个 JSP 网页请求之间，随后此属性就会失效。

- sessionScope

范围和 JSP Scope 中的 session 相同，它的属性范围就是用户持续在服务器连接的时间。

- applicationScope

范围和 JSP Scope 中的 application 相同，它的属性范围是从服务器一开始执行服务，到服务器关闭为止。

6-3-2 与输入有关的隐含对象

与输入有关的隐含对象有两个：param 和 paramValues，它们是 EL 中比较特别的隐含对象。一般而言，我们在取得用户的请求参数时，可以利用下列方法：

```
request.getParameter(String name)
request.getParameterValues(String name)
```

在 EL 中则可以使用 param 和 paramValues 两者来取得数据。

```
${param.name}
${paramValues.name}
```

这里 param 的功能和 `request.getParameter(String name)` 相同，而 paramValues 和 `request.getParameterValues(String name)` 相同。如果用户填了一个表格，表格名称为 username，则我们就可以使用 `${param.username}` 来取得用户填入的值。

为了让读者更加了解 param 和 paramValues 隐含对象的使用，再来看下面这个范例。此范例共有两个文件，分别为给用户输入值用的 *Param.html* 和显示出用户所传之值的 *Param.jsp*。

■ Param.html

```
<html>
<head>
  <title>CH6 - Param.html</title>
</head>
<body>

<h2>EL 隐含对象 param、paramValues</h2>

<form method = "post" action = "Param.jsp">
```

JSP2.0 技术手册

```

<p>姓名: <input type="text" name="username" size="15" /></p>
<p>密码: <input type="password" name="password" size="15" /></p>
<p>性别: <input type="radio" name="sex" value="Male" checked/> 男
        <input type="radio" name="sex" value="Female" /> 女</p>

<p>年龄:
    <select name="old">
        <option value="10">10 - 20</option>
        <option value="20" selected>20 - 30</option>
        <option value="30">30 - 40</option>
        <option value="40">40 - 50</option>
    </select>
</p>

<p>兴趣:
    <input type="checkbox" name="habit" value="Reading" />看书
    <input type="checkbox" name="habit" value="Game" />玩游戏
    <input type="checkbox" name="habit" value="Travel" />旅游
    <input type="checkbox" name="habit" value="Music" />听音乐
    <input type="checkbox" name="habit" value="Tv" />看电视
</p>
<p>
    <input type="submit" value="传送" />
    <input type="reset" value="清除" />
</p>

</form>
</body>
</html>

```

Param.html 的执行结果如图 6-1 所示。当我们把窗体填好后按下传送钮, 它将会把信息传送到 *Param.jsp* 做处理。

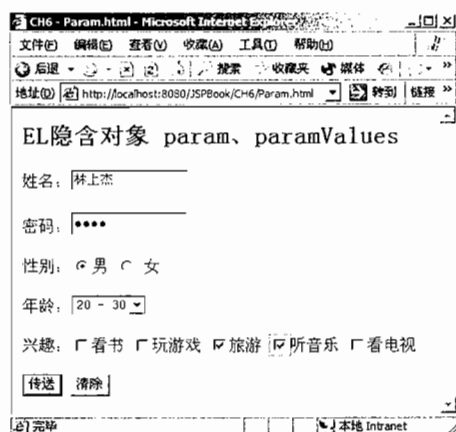


图 6-1 *Param.html* 的执行结果, 并填入信息

接下来, *Param.jsp* 接收由 *Param.html* 传来的信息, 并且将它显示出来:

■ *Param.jsp*

```
<%@ page contentType="text/html; charset=GB2312" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>

<html>
<head>
  <title>CH6 - Param.jsp</title>
</head>
<body>

  <h2>EL 隐含对象 param、paramValues</h2>

  <fmt:requestEncoding value="GB2312" />

  姓名: ${param.username}</br>
  密码: ${param.password}</br>
  性别: ${param.sex}</br>
  年龄: ${param.old}</br>
  兴趣: ${paramValues.habit[0]}
       ${paramValues.habit[1]}

</body>
</html>
```

由 *Param.html* 窗体传过来的值, 我们必须指定编码方式, 才能够确保 *Param.jsp* 能够顺利接收中文, 传统的做法为:

```
<%
  request.setCharacterEncoding("GB2312");
%>
```

假若是使用 JSTL 写法时, 必须使用 I18N 格式处理的标签库, 如下:

```
<fmt:requestEncoding value="GB2312" />
```

Param.jsp 主要使用 EL 的隐含对象 *param* 来接收数据。但是必须注意: 假若要取得多重选择的复选框的值时, 必须使用 *paramValues*, 例如: 使用 *paramValues* 来取得“兴趣”的值, 不过这里笔者最多只显示两笔“兴趣”的值:

```
${param.username}
.....
${paramValues.habit[0]}
${paramValues.habit[1]}
```

有关 JSTL 的使用, 第七章有更加详细的说明。图 6-2 是 *Param.jsp* 的执行结果:

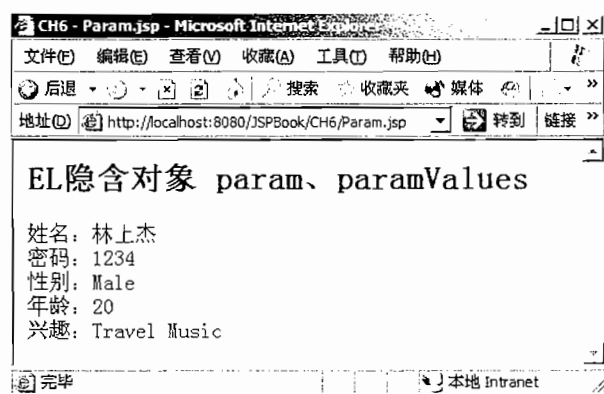


图 6-2 Param.jsp 的执行结果

6-3-3 其他隐含对象

介绍完上面六个隐含对象后，接下来将介绍最后五个隐含对象。

● cookie

所谓的 cookie 是一个小小的文本文件，它是以 key、value 的方式将 Session Tracking 的内容记录在这个文本文件内，这个文本文件通常存在于浏览器的暂存区内。JSTL 并没有提供设定 cookie 的动作，因为这个动作通常都是后端开发者必须去做的事情，而不是交给前端的开发者。假若我们在 cookie 中设定一个名称为 userCountry 的值，那么可以使用 `${cookie.userCountry}` 来取得它。

● header 和 headerValues

header 储存用户浏览器和服务端用来沟通的数据，当用户要求服务端的网页时，会送出一个记载要求信息的标头文件，例如：用户浏览器的版本、用户计算机所设定的区域等其他相关数据。假若要取得用户浏览器的版本，即 `${header["User-Agent"]}`。另外在鲜少机会下，有可能同一标头名称拥有不同的值，此时必须改为使用 headerValues 来取得这些值。

注意

因为 User-Agent 中包含“-”这个特殊字符，所以必须使用“[]”，而不能写成 `$(header.User-Agent)`。

● initParam

就像其他属性一样，我们可以自行设定 web 站台的环境参数(Context)，当我们想取得这些参数时，可以使用 initParam 隐含对象去取得它，例如：当我们在 web.xml 中设定如下：

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
  version="2.4">
```

JSP2.0 技术手册

```

:
    <context-param>
        <param-name>userid</param-name>
        <param-value>mike</param-value>
    </context-param>
:
</web-app>

```

那么我们就可以直接使用 `${initParam.userid}` 来取得名称为 `userid`, 其值为 `mike` 的参数。下面是之前的做法:

```
String userid = (String)application.getInitParameter("userid");
```

● pageContext

我们可以使用 `${pageContext}` 来取得其他有关用户要求或页面的详细信息。表 6-5 列出了几个比较常用的部分。

表 6-5

Expression	说 明
<code>\${pageContext.request.queryString}</code>	取得请求的参数字符串
<code>\${pageContext.request.requestURL}</code>	取得请求的 URL, 但不包括请求之参数字符串
<code>\${pageContext.request.contextPath}</code>	服务的 web application 的名称
<code>\${pageContext.request.method}</code>	取得 HTTP 的方法(GET、POST)
<code>\${pageContext.request.protocol}</code>	取得使用的协议(HTTP/1.1、HTTP/1.0)
<code>\${pageContext.request.remoteUser}</code>	取得用户名称
<code>\${pageContext.request.remoteAddr}</code>	取得用户的 IP 地址
<code>\${pageContext.session.new}</code>	判断 session 是否为新的, 所谓新的 session, 表示刚由 server 产生而 client 尚未使用
<code>\${pageContext.session.id}</code>	取得 session 的 ID
<code>\${pageContext.servletContext.serverInfo}</code>	取得主机端的服务信息

我们来看下面这个范例: `pageContext.jsp`, 相信对读者来说能更加了解 `pageContext` 的用法。

■ pageContext.jsp

```

<%@ page contentType="text/html; charset=GB2312" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<html>
<head>
    <title>CH6 - pageContext.jsp</title>
</head>
<body>

<h2>EL 隐含对象 pageContext</h2>

\${pageContext.request.queryString}:${pageContext.request.queryString}</br>
\${pageContext.request.requestURL}:${pageContext.request.requestURL}</br>
\${pageContext.request.contextPath}:${pageContext.request.contextPath}</br>
\${pageContext.request.method}:${pageContext.request.method}</br>

```

JSP2.0 技术手册

```

\${pageContext.request.protocol}:${pageContext.request.protocol}</br>
\${pageContext.request.remoteUser}:${pageContext.request.remoteUser}</br>
\${pageContext.request.remoteAddr}:${pageContext.request.remoteAddr}</br>
\${pageContext.session.new}:${pageContext.session.new}</br>
\${pageContext.session.id}:${pageContext.session.id}</br>

</body>
</html>

```

`pageContext.jsp` 的执行结果如图 6-3，执行时必须在 `pageContext.jsp` 之后加上 `?test=1234`，即 `PageContext.jsp?test=1234`，这样 `${pageContext.request.queryString}` 才会显示 `test=1234`。

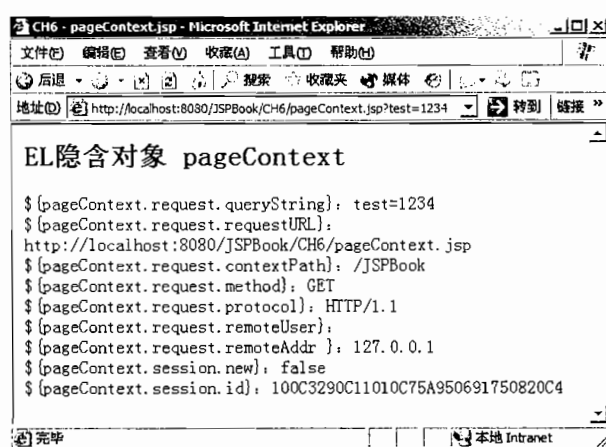


图 6-3 `pageContext.jsp` 的执行结果

注意

因为 `${}` 在 JSP 2.0 中是特殊字符，JSP 容器会自动将它当做 EL 来执行，因此，假若要显示 `${}` 时，必须在 `$` 前加上 `\`，如：`\${ XXXXX }`

6-4 EL 算术运算符

EL 算术运算符主要有以下五个（见表 6-6）：

表 6-6

算术运算符	说 明	范 例	结 果
+	加	<code>\${ 17 + 5 }</code>	22
-	减	<code>\${ 17 - 5 }</code>	12
*	乘	<code>\${ 17 * 5 }</code>	85
/ 或 div	除	<code>\${ 17 / 5 }</code> 或 <code>\${ 17 div 5 }</code>	3
% 或 mod	余数	<code>\${ 17 % 5 }</code> 或 <code>\${ 17 mod 5 }</code>	2

接下来，我们依照下列几种情况，详细说明 EL 算术运算符的规则：

(1) A {+, -, *} B

- 假若 A 和 B 为 null：回传 (Long)0
- 假若 A 或 B 为 BigDecimal 时，将另一个也转为 BigDecimal，则：
 - 假若运算符为 + 时：回传 A.add(B)
 - 假若运算符为 - 时：回传 A.subtract(B)
 - 假若运算符为 * 时：回传 A.multiply(B)
- 假若 A 或 B 为 Float、Double 或包含 e/E 的字符串时：
 - 假若 A 或 B 为 BigInteger 时，将另一个转为 BigDecimal，然后依照运算符执行运算
 - 否则，将两者皆转为 Double，然后依照运算符执行运算
- 假若 A 或 B 为 BigInteger 时，将另一个也转为 BigInteger，则：
 - 假若运算符为 + 时：回传 A.add(B)
 - 假若运算符为 - 时：回传 A.subtract(B)
 - 假若运算符为 * 时：回传 A.multiply(B)
- 否则，将 A 和 B 皆转为 Long，然后依照运算符执行运算
- 假若运算结果产生异常时，则错误！

(2) A {/, div} B

- 假若 A 和 B 为 null：回传 (Long)0
- 假若 A 或 B 为 BigDecimal 或 BigInteger 时，皆转为 BigDecimal，然后回传 A.divide(B, BigDecimal.ROUND_HALF_UP)
- 否则，将 A 和 B 皆转为 Double，然后依照运算符执行运算
- 假若运算结果产生异常时，则错误！

(3) A {% , mod} B

- 假若 A 和 B 为 null：回传 (Long)0
- 假若 A 或 B 为 BigDecimal、Float、Double 或包含 e/E 的字符串时，皆转为 Double，然后依照运算符执行运算
- 假若 A 或 B 为 BigInteger 时，将另一个转为 BigInteger，则回传 A.remainder(B)
- 否则，将 A 和 B 皆转为 Long，然后依照运算符执行运算
- 假若运算结果产生异常时，则错误！

(4) -A

- 假若 A 为 null：回传 (Long)0
- 假若 A 为 BigDecimal 或 BigInteger 时，回传 A.negate()
- 假若 A 为 String 时：
 - 假若 A 包含 e/E 时，将转为 Double，然后依照运算符执行运算
 - 否则，转为 Long，然后依照运算符执行运算
 - 假若运算结果产生异常时，则错误！

- 假若 A 为 Byte、Short、Integer、Long、Float 或 Double
 - 直接依原本类型执行运算
 - 假若运算结果产生异常时，则 错误!
- 否则，错误!

Tomcat 上的 *jsp-examples* 中，有一个 EL 算术运算符的范例 *basic-arithmetic.jsp*。它的程序很简单，所以不在这里多做说明，它的执行结果如图 6-4 所示。

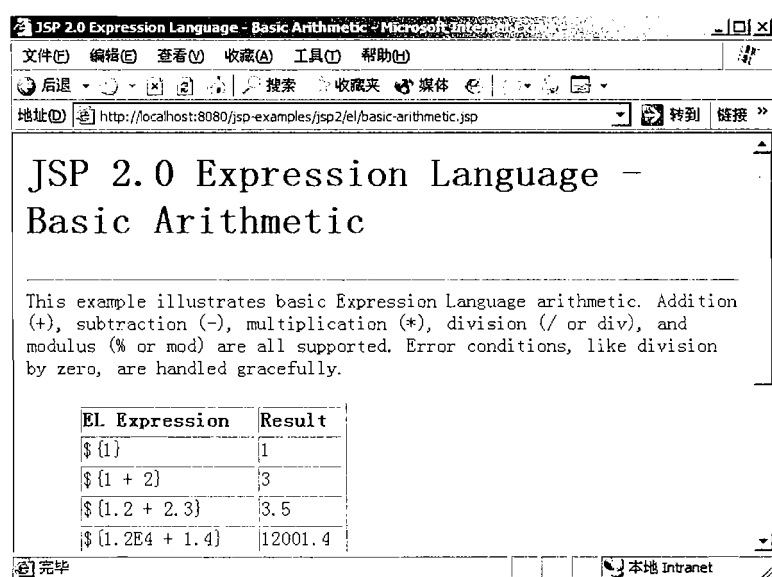


图 6-4 basic-arithmetic.jsp 的执行结果

6-5 EL 关系运算符

EL 关系运算符有以下六个运算符（见表 6-7）：

表 6-7

关系运算符	说 明	范 例	结 果
<code>=</code> 或 <code>eq</code>	等于	<code>\${5 = 5}</code> 或 <code>\${5 eq 5}</code>	true
<code>!=</code> 或 <code>ne</code>	不等于	<code>\${5 != 5}</code> 或 <code>\${5 ne 5}</code>	false
<code><</code> 或 <code>lt</code>	小于	<code>\${3 < 5}</code> 或 <code>\${3 lt 5}</code>	true
<code>></code> 或 <code>gt</code>	大于	<code>\${3 > 5}</code> 或 <code>\${3 gt 5}</code>	false
<code><=</code> 或 <code>le</code>	小于等于	<code>\${3 <= 5}</code> 或 <code>\${3 le 5}</code>	true
<code>>=</code> 或 <code>ge</code>	大于等于	<code>\${3 >= 5}</code> 或 <code>\${3 ge 5}</code>	false

注意

在使用 EL 关系运算符时，不能够写成：

```
${param.password1} == ${param.password2}
```

或者

```
${ ${param.password1} == ${param.password2} }
```

而应写成

```
${ param.password1 == param.password2 }
```

接下来，我们依照下列几种情况，详细说明 EL 关系运算符的规则：

(1) A {<, >, <=, >=, lt, gt, le, ge} B

- 假若 A = B，运算符为 <=, le, >=, ge 时，回传 true，否则回传 false
- 假若 A 为 null 或 B 为 null 时，回传 false
- 假若 A 或 B 为 BigDecimal 时，将另一个转为 BigDecimal，然后回传 A.compareTo(B) 的值
- 假若 A 或 B 为 Float、Double 时，皆转为 Double 类型，然后依其运算符运算
- 假若 A 或 B 为 BigInteger 时，将另一个转为 BigInteger，然后回传 A.compareTo(B) 的值
- 假若 A 或 B 为 Byte、Short、Character、Integer 或 Long 时，皆转为 Long 类型，然后依其运算符运算
- 假若 A 或 B 为 String 时，将另一个也转为 String，然后做词汇上的比较
- 假若 A 为 Comparable 时，则：
 - 假若 A.compareTo(B) 产生异常时，则错误！
- 否则，采用 A.compareTo(B) 的比较结果
- 假若 B 为 Comparable 时，则：
 - 假若 B.compareTo(A) 产生异常时，则错误！
- 否则，采用 A.compareTo(B) 的比较结果
- 否则，错误！

(2) A {=, !=, eq, ne} B

- 假若 A = B，依其运算符运算
- 假若 A 为 null 或 B 为 null 时：= / eq 则回传 false，!= / ne 则回传 true
- 假若 A 或 B 为 BigDecimal 时，将另一个转为 BigDecimal，则：
 - 假若运算符为 = / eq，则 回传 A.equals(B)
 - 假若运算符为 != / ne，则 回传 !A.equals(B)
- 假若 A 或 B 为 Float、Double 时，皆转为 Double 类型，然后依其运算符运算
- 假若 A 或 B 为 BigInteger 时，将另一个转为 BigInteger，则：
 - 假若运算符为 = / eq，则 回传 A.equals(B)
 - 假若运算符为 != / ne，则 回传 !A.equals(B)
- 假若 A 或 B 为 Byte、Short、Character、Integer 或 Long 时，皆转为 Long 类型，然后依其运算符运算

- 假若 A 或 B 为 Boolean 时，将另一个也转为 Boolean，然后依其运算符运算
- 假若 A 或 B 为 String 时，将另一个也转为 String，然后做词汇上的比较
- 否则，假若 A.equals(B)产生异常时，则 错误!
- 否则，然后依其运算符运算，回传 A.equals(B)

Tomcat 上的 jsp-examples 中，有一个 EL 关系运算符的范例 *basic-comparisons.jsp*。它的程序很简单，所以不在这里多做说明，大家直接看它的执行结果（如图 6-5 所示）：

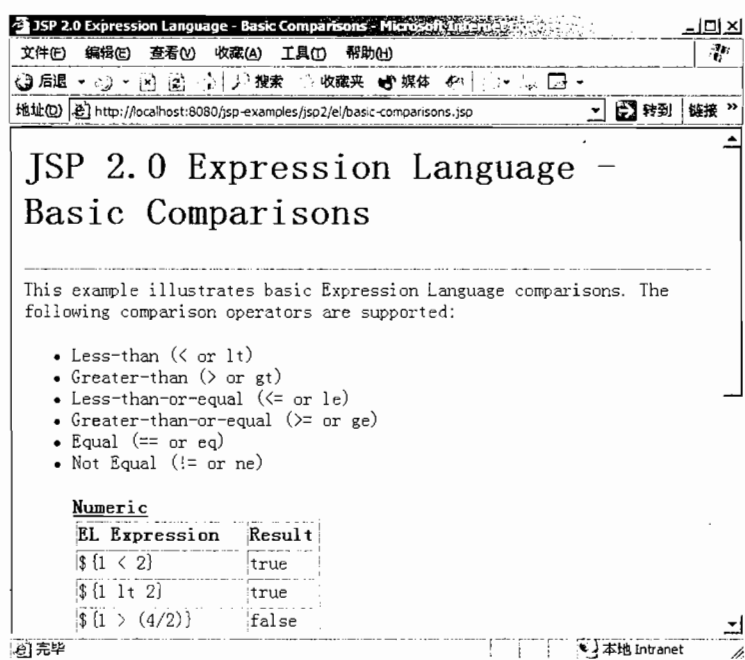


图 6-5 basic-comparisons.jsp 的执行结果

6-6 EL 逻辑运算符

EL 逻辑运算符只有三个（见表 6-8）：

表 6-8

逻辑运算符	说 明	范 例	结 果
&& 或 and	交集	<code>\${A && B}</code> 或 <code>\${A and B}</code>	true / false
或 or	并集	<code>\${A B}</code> 或 <code>\${A or B}</code>	true / false
! 或 not	非	<code>\${!A}</code> 或 <code>\${not A}</code>	true / false

下面举几个例子：

```
${ param.month == 7 and param.day == 14 }  
${ param.month == 7 || param.day == 14 }  
${ not param.choice }
```

EL 逻辑运算符的规则很简单:

- (1) A {&&, and, || 或 or } B
 - 将 A 和 B 转为 Boolean, 然后依其运算符运算
- (2) {!, not}A
 - 将 A 转为 Boolean, 然后依其运算符运算

6-7 EL 其他运算符

EL 除了上述三大类的运算符之外, 还有下列几个重要的运算符:

- (1) Empty 运算符
- (2) 条件运算符
- (3) () 括号运算符

6-7-1 Empty 运算符

Empty 运算符主要用来判断值是否为 null 或空的, 例如:

```
${ empty param.name }
```

接下来说明 Empty 运算符的规则:

- (1) {empty} A
 - 假若 A 为 null 时, 回传 true
 - 否则, 假若 A 为空 String 时, 回传 true
 - 否则, 假若 A 为空 Array 时, 回传 true
 - 否则, 假若 A 为空 Map 时, 回传 true
 - 否则, 假若 A 为空 Collection 时, 回传 true
 - 否则, 回传 false

6-7-2 条件运算符

所谓条件运算符如下:

```
${ A ? B : C }
```

意思是说, 当 A 为 true 时, 执行 B; 而 A 为 false 时, 则执行 C。

6-7-3 括号运算符

括号运算符主要用来改变执行优先权，例如：\${ A * (B+C) }

至于运算符的优先权，如下所示(由高至低，由左至右)：

- [], .
- ()
- -(负)、not、!、empty
- *, /、div、%、mod
- +、-(减)
- <、>、<=、>=、lt、gt、le、ge
- ==、!=、eq、ne
- &&、and
- ||、or
- \${ A ? B : C }

最后笔者写一个 *ELOperator.jsp* 范例，将所有运算符实际操作一遍。

■ *ELOperator.jsp*

```
<%@ page contentType="text/html; charset=GB2312" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<html>
<head>
<title>CH6 - ELOperator.jsp</title>
</head>
<body>

<h2>EL 的运算符</h2>

<c:set value="mike" var="username" scope="request" />
<table border="1" width="50%" align="left">
<TR>
<TR>
<TH>运算式</TH>
<TH>结果</TH>
</TR>
<TR><TD>14 + 3</TD><TD>${14 + 3}</TD></TR>
<TR><TD>14 - 3</TD><TD>${14 - 3}</TD></TR>
<TR><TD>14 * 3</TD><TD>${14 * 3}</TD></TR>
<TR><TD>14 / 3</TD><TD>${14 / 3}</TD></TR>
<TR><TD>14 % 3</TD><TD>${14 % 3}</TD></TR>
<TR><TD>14 == 3</TD><TD>${14 == 3}</TD></TR>
<TR><TD>14 != 3</TD><TD>${14 != 3}</TD></TR>
<TR><TD>14 < 3</TD><TD>${14 < 3}</TD></TR>
<TR><TD>14 > 3</TD><TD>${14 > 3}</TD></TR>
<TR><TD>14 <= 3</TD><TD>${14 <= 3}</TD></TR>
```

JSP2.0 技术手册

```

<TR><TD>14 >= 3</TD><TD>${14 >= 3}</TD></TR>
<TR><TD>true && false</TD><TD>${true && false}</TD></TR>
<TR><TD>true || false</TD><TD>${true || false}</TD></TR>
<TR><TD>! false</TD><TD>${! false}</TD></TR>
<TR><TD>empty username</TD><TD>${empty username}</TD></TR>
<TR><TD>empty password</TD><TD>${empty password}</TD></TR>
</table>
</body>
</html>

```

EL 的数学运算符、相等运算符、关系运算符和逻辑运算符就跟其他程序语言一样，并没有特别的地方。但是它的 `empty` 运算符就比较特别，为了测试它，笔者写了这样一行程序代码：

```
<c:set value="mike" var="username" scope="request" />
```

这样 Request 属性范围里就存在一个名称为 `username`、值为 `mike` 的属性。执行此程序时，读者将会发现 `${empty username}` 为 `false`；`${empty password}` 为 `true`，其代表的意义就是：它可以在四种属性范围中找到 `username` 这个属性，但是找不到 `password` 这个属性。`ELOperator.jsp` 的执行结果如图 6-6：

运算式	结果
14 + 3	17
14 - 3	11
14 * 3	42
14 / 3	4.666666666666667
14 % 3	2
14 == 3	false
14 != 3	true
14 < 3	false
14 > 3	true
14 <= 3	false
14 >= 3	true
true && false	false
true false	true
! false	true
empty username	false
empty password	true

图 6-6 ELOperator.jsp 的执行结果

6-8 EL Functions

前面几节主要介绍 EL 语法的使用和规则，本节笔者将介绍如何自定义 EL 的函数(functions)。

JSP2.0 技术手册

EL 函数的语法如下:

```
ns:function( arg1, arg2, arg3 ... argN)
```

其中 ns 为前置名称(prefix), 它必须和 taglib 指令的前置名称一样。如下范例:

```
<% @ taglib prefix="my"
    uri="http://jakarta.apache.org/tomcat/jsp2-example-taglib" %>
.....
${my:function(param.name)}
```

前置名称都为 my, 至于 function 为 EL 函数的名称, 而 arg1、arg2 等等, 都是 function 的传入值。在 Tomcat 5.0.16 中有一个简单的 EL 函数范例, 名称为 *functions.jsp*, 笔者接下来将依此范例来说明如何自定义 EL 函数。

6-8-1 Tomcat EL 函数范例

Tomcat 提供的 EL 函数范例中, 自定义两个 EL 函数: reverse 和 countVowels, 其中:

reverse 函数: 将传入的字符串以反向顺序输出。

countVowels 函数: 计算传入的字符串中, 和 aeiouAEIOU 吻合的字符个数。

图 6-7 是 *functions.jsp* 程序的执行结果:

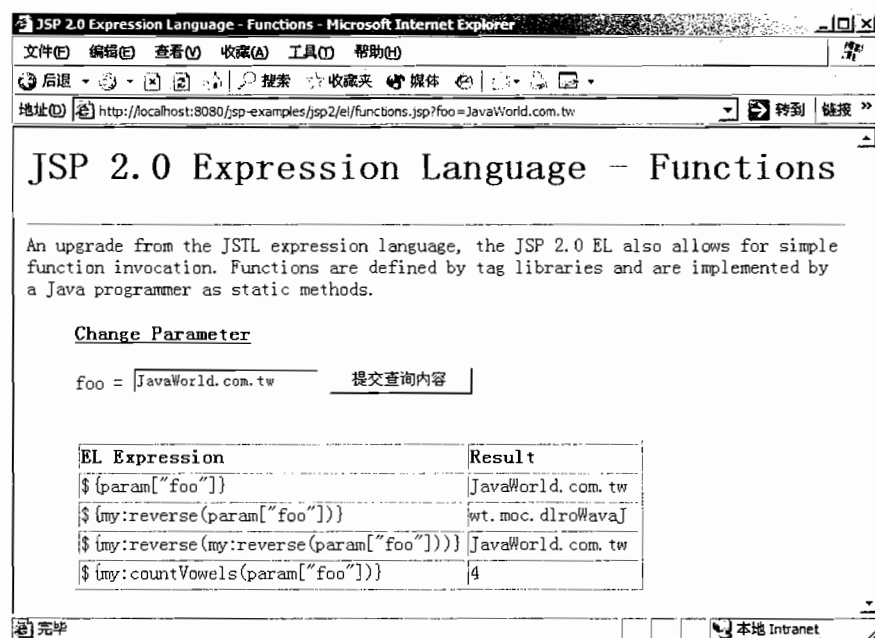


图 6-7 functions.jsp 的执行结果

输入 JavaWorld.com.tw 字符串至 reverse 函数后, 回传 wt.moc.dlroWavaJ 的结果; 若

传入 countVowels 函数后，因为有两个 a 和 o，总共四个字符吻合，所以回传 4。

Tomcat 的 EL 函数范例，主要分为四个部分（见表 6-9）：

表 6-9

<i>web.xml</i>	设定 taglib 的 TLD 文件位置
<i>functions.jsp</i>	使用 EL 函数的范例程序
<i>jsp2-example-taglib.tld</i>	EL 函数、标签库的设定文件
<i>jsp2.examples.el.Functions.java</i>	EL 函数主要程序逻辑处理部分

这四个部分环环相扣，都互有关系，笔者依 *functions.jsp* 为中心，然后再慢慢说明其他部分。首先我们直接来看 *functions.jsp* 程序：

6-8-2 functions.jsp

■ functions.jsp

```
<%@ taglib prefix="my"
uri="http://jakarta.apache.org/tomcat/jsp2-example-taglib"%>
<html>
  <head>
    <title>JSP 2.0 Expression Language - Functions</title>
  </head>
  <body>
    <h1>JSP 2.0 Expression Language - Functions</h1>
    ... 略
    <blockquote>
      <u><b>Change Parameter</b></u>
      <form action="functions.jsp" method="GET">
        foo = <input type="text" name="foo" value="\${param['foo']}">
        <input type="submit">
      </form>
      <br>
      <code>
        <table border="1">
          <thead>
            <td><b>EL Expression</b></td>
            <td><b>Result</b></td>
          </thead>
          <tr>
            <td>\${param["foo"]}</td>
            <td>\${param["foo"]}&nbsp;</td>
          </tr>
          <tr>
            <td>\${my:reverse(param["foo"])}</td>
            <td>\${my:reverse(param["foo"])}&nbsp;</td>
          </tr>
          <tr>
            <td>\${my:reverse(my:reverse(param["foo"]))}</td>
          </tr>
        </table>
      </code>
    </blockquote>
  </body>
</html>
```

JSP2.0 技术手册


```

        <td>${my:reverse(my:reverse(param["foo"]))}&nbsp;</td>
    </tr>
    <tr>
        <td>\${my:countVowels(param["foo"])}</td>
        <td>${my:countVowels(param["foo"])}&nbsp;</td>
    </tr>
</table>
</code>
</blockquote>
</body>
</html>

```

functions.jsp 程序中,一开始定义 taglib,它的前置名称为 my; uri 为 <http://jakarta.apache.org/tomcat/jsp2-example-taglib>,如下所示:

```

<%@ taglib prefix="my"
uri="http://jakarta.apache.org/tomcat/jsp2-example-taglib"%>

```

当 Container 执行这段程序时,它会根据 uri 的值,到 *web.xml* 中找相对应的 TLD (Tag Library Descriptor) 文件。至于 *web.xml* 如何设定两者之间的对应关系,我们在 6-8-3 小节再说明。

functions.jsp 中包含一个窗体(form),当用户在文本输入框(text input)中输入字符串,按下按钮时,底下会显示字符串经过 EL 函数处理后的结果。*functions.jsp* 程序最重要的部分是调用 EL 函数:

```

${my:reverse(param["foo"])}

```

上述的意思是接收 foo 参数,然后传入 reverse 函数。调用 EL 函数的方式很简单,只要前置名称:其中 EL 函数名称是被定义在 TLD 文件中,这会在 6-8-4 小节详细说明。至于 reverse 函数的逻辑运算,则是被定义在 *jsp2.examples.el.Functions.java* 程序中,这部分会在 6-8-5 小节中说明。

注意

TLD 文件主要为标签的设定文件,其中包含标签的名称、参数等等。在 JSP 2.0 之后,相关 EL 函数的设定,也可以在 TLD 文件中定义。

6-8-3 web.xml

web.xml 是每个 web 站台最主要的设定文件,在这个设定文件中,可以设定许多东西,如:Servlet、Resource、Filter 等等。不过现在关心的是如何在 *web.xml* 中设定 taglib 的 uri 是对应到哪个 TLD 文件。笔者从范例的 *web.xml* 中节录出设定的片段程序如下:

■ web.xml

```

<jsp-config>
    <taglib>
        <taglib-uri>
            http://jakarta.apache.org/tomcat/jsp2-example-taglib
        </taglib-uri>
        <taglib-location>

```

JSP2.0 技术手册

```

    /WEB-INF/jsp2/jsp2-example-taglib.tld
  </taglib-location>
</taglib>
</jsp-config>

```

在 *web.xml* 中, <taglib>用来设定标签的 TLD 文件位置。<taglib-uri>用来指定 taglib 的 uri 位置, 用户可以自行给定一个 uri, 例如:

```

<taglib-uri>http://www.javaworld.com.tw/jute</taglib-uri>
<taglib-uri>tw.com.javaworld</taglib-uri>

```

<taglib-location>用来指定 TLD 文件的位置。依照范例, 它是指定在 *WEB-INF/jsp2/*目录下的 *jsp2-example-taglib.tld*。

因此, 笔者所节录下来的 *web.xml*, 它所代表的意思是: taglib 的 uri 为 *http://jakarta.apache.org/tomcat/jsp2-example-taglib*, 它的 TLD 文件是在 *WEB-INF/jsp2/*目录下的 *jsp2-example-taglib.tld*。

6-8-4 jsp2-example-taglib.tld

在 *jsp2-example-taglib.tld* 中定义许多标签, 其中笔者节录一段定义 EL 函数:

■ jsp2-example-taglib.tld

```

<function>
  <description>Reverses the characters in the given String</description>
  <name>reverse</name>
  <function-class>jsp2.examples.el.Functions</function-class>
  <function-signature>
    java.lang.String reverse( java.lang.String )
  </function-signature>
</function>
<function>
  <description>Counts the number of vowels (a,e,i,o,u) in the given
    String</description>
  <name>countVowels</name>
  <function-class>jsp2.examples.el.Functions</function-class>
  <function-signature>
    java.lang.String numVowels( java.lang.String )
  </function-signature>
</function>

```

上述定义两个 EL 函数, 用<name>来设定 EL 函数名称, 它们分别为 *reverse* 和 *countVowels*; 用<function-class>设定 EL 函数的 Java 类, 本范例的 EL 函数都是定义在 *jsp2.examples.el.Functions*; 最后用<function-signature>来设定 EL 函数的传入值和回传值, 例如:

```

<function-signature>java.lang.String
  reverse( java.lang.String )</function-signature>

```

表示 *reverse* 函数有一 *String* 类型的传入值, 然后回传 *String* 类型的值。最后我们再来看

reverse 和 countVowels 的程序。

6-8-5 Functions.java

Functions.java 主要定义三个公开静态的方法，分别为：reverse、numVowels 和 caps（见表 6-10）。下面是 *Functions.java* 完整的程序代码：

■ *Functions.java*

```
package jsp2.examples.el;

import java.util.*;

/**
 * Defines the functions for the jsp2 example tag library.
 * <p>Each function is defined as a static method.</p>
 */
public class Functions {
    public static String reverse( String text ) {
        return new StringBuffer( text ).reverse().toString();
    }

    public static int numVowels( String text ) {
        String vowels = "aeiouAEIOU";
        int result = 0;
        for( int i = 0; i < text.length(); i++ ) {
            if( vowels.indexOf( text.charAt( i ) ) != -1 ) {
                result++;
            }
        }
        return result;
    }

    public static String caps( String text ) {
        return text.toUpperCase();
    }
}
```

表 6-10

String reverse(String text)	将 text 字符串的顺序反向处理，然后回传反向后的字符串
int numVowels(String text)	将 text 字符串比对 aeiouAEIOU 等字符，然后回传比对中的次数
String caps(String text)	将 text 字符串都转为大写，然后回传此字符串

注意

在定义 EL 函数时，都必须为公开静态(public static)

7

第七章

JSTL 1.1

JSTL 全名为 JavaServer Pages Standard Tag Library，目前最新的版本为 1.1 版。JSTL 是由 JCP(Java Community Process)所制定的标准规范，它主要提供给 Java Web 开发人员一个标准通用的标签函数库。

Web 程序员能够利用 JSTL 和 EL 来开发 Web 程序，取代传统直接在页面上嵌入 Java 程序(Scripting)的做法，以提高程序的阅读性、维护性和方便性。

本章中，笔者将详细介绍如何使用 JSTL 中各种不同的标签，将依序介绍条件、循环、URL、U18N、XML、SQL 等标签的用法，让读者对 JSTL 有更深层的了解，并且能够学会如何使用 JSTL。本章将分 6 节来介绍：

- 7-1 JSTL 1.1 简介
- 7-2 核心标签库 (Core tag library)
- 7-3 I18N 格式标签库 (I18N-capable formatting tags library)
- 7-4 SQL 标签库 (SQL tag library)
- 7-5 XML 标签库 (XML tag library)
- 7-6 函数标签库 (Functions tag library)

JSP2.0 技术手册

7-1 JSTL 1.1 简介

JavaServer Pages Standard Tag Library (1.1), 它的中文名称为 JSP 标准标签函数库。JSTL 是一个标准的已制定好的标签库, 可以应用于各种领域, 如: 基本输入输出、流程控制、循环、XML 文件剖析、数据库查询及国际化和文字格式标准化的应用等。从表 7-1 可以知道, JSTL 所提供的标签函数库主要分为五大类:

- (1) 核心标签库 (Core tag library)
- (2) I18N 格式标签库 (I18N-capable formatting tag library)
- (3) SQL 标签库 (SQL tag library)
- (4) XML 标签库 (XML tag library)
- (5) 函数标签库 (Functions tag library)

表 7-1

JSTL	前置名称	URI	范 例
核心标签库	c	http://java.sun.com/jsp/jstl/core	<c:out>
I18N 格式标签库	fmt	http://java.sun.com/jsp/jstl/xml	<fmt:formatDate>
SQL 标签库	sql	http://java.sun.com/jsp/jstl/sql	<sql:query>
XML 标签库	xml	http://java.sun.com/jsp/jstl/fmt	<x:forEach>
函数标签库	fn	http://java.sun.com/jsp/jstl/functions	<fn:split>

另外, JSTL 也支持 EL(Expression Language)语法, 例如: 在一个标准的 JSP 页面中可能会使用到如下的写法:

```
<%= userList.getUser().getPhoneNumber() %>
```

使用 JSTL 搭配传统写法会变成这样:

```
<c_rt:out value="<%= userList.getUser().getPhoneNumber() %>" />
```

使用 JSTL 搭配 EL, 则可以改写成如下的形式:

```
<c:out value="${userList.user.phoneNumber}" />
```

虽然对网页设计者来说, 假如没有学过 Java Script 或者是第一次看到这种写法时, 可能会搞不太懂, 但是与 Java 语法相比, 这应该更容易学习。

7-1-1 安装使用 JSTL 1.1

JSTL 1.1 必须在支持 Servlet 2.4 且 JSP 2.0 以上版本的 Container 才可使用。JSTL 主要由 Apache 组织的 Jakarta Project 所实现, 因此读者可以到 <http://jakarta.apache.org/builds/jakarta-taglibs/releases/standard/> 下载实现好的 JSTL 1.1, 或者直接使用本书光盘中 JSTL 1.1, 软件名称为: *jakarta-taglibs-standard-current.zip*。

下载完后解压缩, 可以发现文件夹中所包含的内容如图 7-1 所示:

JSP2.0 技术手册

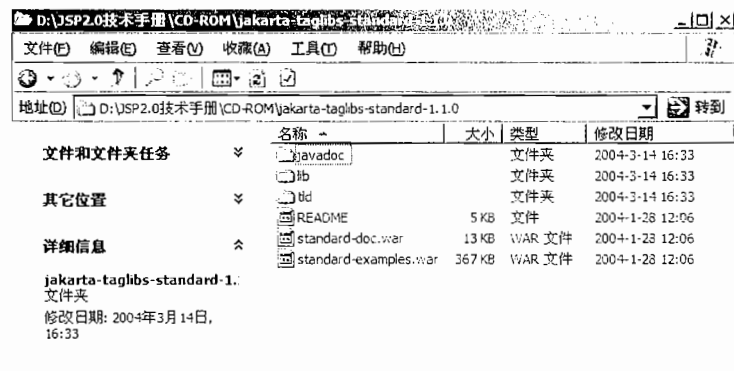


图 7-1 jakarta-taglibs-standard-1.1.0-B1 的目录结构

将 lib 中的 *jstl.jar*、*standard.jar* 复制到 Tomcat 的 *WEB-INF/lib* 中，然后就可以在 JSP 网页中使用 JSTL 了。除了复制 *.jar* 文件外，最好也把 *tld* 文件的目录也复制到 *WEB-INF* 中，以便日后使用。

注意

lib 目录下，除了 *jstl.jar* 和 *standard.jar* 之外，还有 *old-dependencies* 目录，这目录里面的东西是让之前 JSTL 1.0 的程序也能够在此 JSTL 1.1 环境下使用。*tld* 目录下有许多 TLD 文件，其中大部分都是 JSTL 1.0 的 TLD 文件，例如：*c-1_0.tld* 和 *c-1_0-rt.tld*。

下面写一个测试用的范例程序 *HelloJSTL.jsp*，程序主要是显示浏览器的版本和欢迎的字符串。

■ HelloJSTL.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<html>
<head>
<title>测试你的第一个使用到 JSTL 的网页</title>
</head>

<body>
<c:out value="欢迎测试你的第一个使用到 JSTL 的网页"/>
<br>你使用的浏览器是: </br>
<c:out value="${header['User-Agent']}" />
<c:set var="a" value="David O'Davies" />
<c:out value="David O'Davies" escapeXml="true" />
</body>
</html>
```

在 *HelloJSTL.jsp* 的范例里，笔者用到核心标签库(Core)中的标准输出功能和 EL 的 *header* 隐含对象。若要在 JSP 网页中使用 JSTL 时，一定要先做下面这行声明：

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

这段声明表示我将使用 JSTL 的核心标签库。一般而言，核心标签库的前置名称(prefix)都

为 c, 当然你也可以自行设定。不过 uri 此时就必须为 `http://java.sun.com/jsp/jstl/core`。

注意

JSTL 1.0 中, 核心标签库的 uri 默认为 `http://java.sun.com/jstl/core`, 比 JSTL 1.1 少一个 `jsp/` 的路径。因为 JSTL 1.1 同时支持 JSTL 1.0 和 1.1, 所以假若核心标签库的 uri 为 `http://java.sun.com/jstl/core`, 则将会使用到 JSTL 1.0 的核心标签库。

接下来使用核心标签库中的 `out` 标签, 显示 value 的值。`${header['User-Agent']}` 表示取得表头里的 `User-Agent` 的值, 即有关用户浏览器的种类。

```
<c:out value="欢迎测试你的第一个使用到 JSTL 的网页" />
<c:out value="${header['User-Agent']}" />
```

`HelloJSTL.jsp` 的执行结果如图 7-2 所示。



图 7-2 `HelloJSTL.jsp` 的执行结果

假若读者想要自定义 taglib 的 uri 时, 那就必须在 `web.xml` 中加入设定值。例如: 假若 uri 想要改为 `http://www.javaworld.com.tw/jstl/core` 时, `web.xml` 就必须加入如下设定:

```
<web-app>
:
  <jsp-config>
    <taglib>
      <taglib-uri>http://www.javaworld.com.tw/jstl/core</taglib-uri>
      <taglib-location>/WEB-INF/tld/c.tld</taglib-location>
    </taglib>
  </jsp-config>
:
</web-app>
```

在上面的设定中, `<taglib-uri>` 主要是设定标签库的 URI; 而 `<taglib-location>` 则是用来设定标签对应的 TLD 文件。因此, 使用 `<%@ taglib %>` 指令时, 可以直接写成如下语句:

```
<%@ taglib prefix="c" uri="http://www.javaworld.com.tw/jsp/jstl/core" %>
```

7-1-2 JSTL 1.1 VS. JSTL 1.0

JSTL 1.0 更新至 JSTL 1.1 时, 有以下几点不同:

JSP2.0 技术手册

(1) EL 原本是定义在 JSTL 1.0 的, 现在 EL 已经正式纳入 JSP 2.0 标准规范中, 所以在 JSTL 1.1 规范中, 已经没有 EL 的部分, 但是 JSTL 依旧能使用 EL。

(2) JSTL 1.0 中, 又分 EL 和 RT 两种函数库, 到了 JSTL 1.1 之后, 已经不再分这两种了。以下说明 EL 和 RT 的差别:

EL

- 完全使用 Expression Language
- 简单
- 建议使用

RT

- 使用 Scriptlet
- Java 语法
- 供不想转换且习惯旧表示法的开发者使用

笔者在此强烈建议大家使用 EL 来做, 简单又方便。

(3) JSTL 1.1 新增函数(functions)标签库, 主要提供一些好用的字符串处理函数, 例如:

fn:contains、fn:containsIgnoreCase、fn:endsWith、fn:indexOf、fn:join、fn:length、fn:replace、fn:split、fn:startsWith 和 fn:substring 等等。

除了上述三项比较大的改变之外, 还包括许多小改变, 在此不多加说明, 有兴趣的读者可以去看 JSTL 1.1 附录 B “Changes” 部分, 那里有更详尽的说明。

7-1-3 安装 standard-examples

当解压缩 *jakarta-taglibs-standard-current.zip* 后, 文件夹内(见图 7-1)有一个 *standard-examples.war* 的文件, 将它移至 Tomcat 的 *webapps* 后, 重新启动 Tomcat 会发现, 在 *webapps* 目录下多了一个 *standard-examples* 的目录。接下来我们打开 IE, 在 URL 位置上输入 <http://localhost:8080/standard-examples>, 你将会看到图 7-3 所示的画面。

这个站台有很多 JSTL 的范例, 它包括以下几部分:

- General Purpose Tags
- Conditional Tags
- Iterator Tags
- Import Tags
- I18N & Formatting Tags
- XML Tags
- SQL Tags
- Functions
- Tag Library Validators
- Miscellaneous



图 7-3 standard-examples 站台

这些范例程序几乎涵盖了所有的 JSTL 标签函数库，假若读者对哪一个标签的使用有问题，可以先来找一找这里的范例程序，应该或多或少会有所帮助。

7-2 核心标签库 (Core tag library)

首先介绍的核心标签库(Core)主要有：基本输入输出、流程控制、迭代操作和 URL 操作。详细的分类如表 7-2 所示，接下来笔者将为读者一一介绍每个标签的功能。

表 7-2

分 类	功能分类	标签名称
Core	表达式操作	out set remove catch
	流程控制	if choose when otherwise
	迭代操作	forEach forTokens

续表

分 类	功能分类	标签名称
Core	URL 操作	import param url param redirect param

在 JSP 中要使用 JSTL 中的核心标签库时, 必须使用 `<%@ taglib %>` 指令, 并且设定 `prefix` 和 `uri` 的值, 通常设定如下:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

上述的功用在于声明将使用 JSTL 的核心标签库。

注意
假若没有上述声明指令, 将无法使用 JSTL 的核心功能, 这是读者在使用 JSTL 时必须要小心地方。

7-2-1 表达式操作

表达式操作分类中包含四个标签: `<c:out>`、`<c:set>`、`<c:remove>`和`<c:catch>`。接下来将依序介绍这四个标签的用法。

● `<c:out>`

`<c:out>`主要用来显示数据的内容, 就像是 `<%= scripting-language %>` 一样, 例如:

```
Hello ! <c:out value="${username}" />
```

语法

语法1: 没有本体(body)内容

```
<c:out value="value" [escapeXml="{true|false}"] [default="defaultValue"] />
```

语法2: 有本体内容

```
<c:out value="value" [escapeXml="{true|false}"]>
default value
</c:out>
```

属性

名 称	说 明	EL	类 型	必 须	默认值
value	需要显示出来的值	Y	Object	是	无
default	如果 value 的值为 null, 则显示 default 的值	Y	Object	否	无
escapeXml	是否转换特殊字符, 如: < 转换成 <	Y	boolean	否	true

JSP2.0 技术手册

注意

表格中的 EL 字段，表示此属性的值是否可以为 EL 表达式，例如：Y 表示 attribute = "\${表达式}" 为符合语法的，N 则反之。

Null 和 错误处理

- 假若 value 为 null，会显示 default 的值；假若没有设定 default 的值，则会显示一个空的字符串。

说明

一般来说，<c:out>默认会将 <、>、'、" 和 & 转换为 <、>、'、" 和 &。假若不想转换时，只需要设定<c:out>的 escapeXml 属性为 false 就可以了（见表 7-3）。

表 7-3

字符	Entity
<	<
>	>
'	'
"	"
&	&

范例

```
<c:out value="Hello JSP 2.0 !! " />
<c:out value="${ 3 + 5 }" />
<c:out value="${ param.data }" default="No Data" />
<c:out value="<p>有特殊字符</p>" />
<c:out value="<p>有特殊字符</p>" escapeXml="false" />
```

1. 在网页上显示 Hello JSP 2.0 !! ；
2. 在网页上显示 8；
3. 在网页上显示由窗体传送过来的 data 参数之值，假若没有 data 参数，或 data 参数的值为 null 时，则网页上会显示 No Data；
4. 在网页上显示 “<p>有特殊字符</p>”；
5. 在网页上显示 “有特殊字符”。

● **<c:set>**

<c:set>主要用来将变量储存至 JSP 范围中或是 JavaBean 的属性中。

语法

语法1：将 value 的值储存至范围为scope的 varName 变量之中

```
<c:set value="value" var="varName"
[scope="{ page|request|session|application }"]/>
```

语法2：将本体内容的数据储存至范围为scope的 varName 变量之中

```
<c:set var="varName" [scope="{ page|request|session|application }"]>
... 本体内容
</c:set>
```

JSP2.0 技术手册

语法3: 将 value 的值储存至 target 对象的属性中

```
<c:set value="value" target="target" property="propertyName" />
```

语法4: 将本体内容的数据储存至 target 对象的属性中

```
<c:set target="target" property="propertyName">
... 本体内容
</c:set>
```

属性

名 称	说 明	EL	类型	必须	默认值
value	要被储存的值	Y	Object	否	无
var	欲存入的变量名称	N	String	否	无
scope	var 变量的 JSP 范围	N	String	否	page
target	为一 JavaBean 或 java.util.Map 对象	Y	Object	否	无
property	指定 target 对象的属性	Y	String	否	无

Null 和 错误处理

- 语法 3 和语法 4 会产生异常错误, 有以下两种情况:
 - target 为 null
 - target 不是 java.util.Map 或 JavaBean 对象
- 假若 value 为 null 时: 将由储存变量改为移除变量
 - 语法 1: 由 var 和 scope 所定义的变量, 将被移除
 - 若 scope 已指定时, 则 `PageContext.removeAttribute(varName, scope)`
 - 若 scope 未指定时, 则 `PageContext.removeAttribute(varName)`
 - 语法 3:
 - 假若 target 为 Map 时, 则 `Map.remove(property)`
 - 假若 target 为 JavaBean 时, property 指定的属性为 null

说明

使用 `<c:set>` 时, var 主要用来存放表达式的结果; scope 则是用来设定储存的范围, 例如: 假若 `scope="session"`, 则将会把数据储存在 session 中。如果 `<c:set>` 中没有指定 scope 时, 则它会默认存在 Page 范围里。

注意
var 和 scope 这两个属性不能使用表达式来表示, 例如: 我们不能写成 <code>scope="\${ourScope}"</code> 或者是 <code>var="\${username}"</code> 。

我们考虑下列的写法:

```
<c:set var="number" scope="session" value="${1 + 1}" />
```

把 1+1 的结果 2 储存到 number 变量中。如果<c:set>没有 value 属性,此时 value 之值在<c:set>和</c:set>之间, 本体内内容看下面的范例:

```
<c:set var="number" scope="session"> .....  
<c:out value="\${1+1}" />  
</c:set>
```

上面的 <c:out value="\\${1+1}" /> 部分可以改写成 2 或是 <%=1+1%>, 结果都会一样, 也就是说, <c:set>是把本体(body)运算后的结果来当做 value 的值。另外, <c:set>会把 body 中最开头和结尾的空白部分去掉。如:

```
<c:set var="number" scope="session">  
    1 + 1  
</c:set>
```

则 number 中储存的值为 1 + 1 而不是 1 + 1。

范例

```
<c:set var="number" scope="request" value="\${1 + 1}" />  
<c:set var="number" scope="session" />  
\${3 + 5}  
</c:set>  
<c:set var="number" scope="request" value="\${ param.number }" />  
<c:set target="User" property="name" value="\${ param.Username}" />
```

1. 将 2 存入 Request 范围的 number 变量中;
2. 将 8 存入 Session 范围的 number 变量中;
3. 假若 \${param.number} 为 null 时,则**移除** Request 范围的 number 变量;若 \${param.number} 不为 null 时, 则将 \${param.number} 的值存入 Request 范围的 number 变量中;
4. 假若 \${param.Username} 为 null 时, 则设定 User(JavaBean)的 name 属性为 null; 若不为 null 时, 则将 \${param.Username} 的值存入 User(JavaBean)的 name 属性(setter 机制)。

注意

上述范例的 3. 中, 假若 \${param.number} 为 null 时, 则表示**移除** Request 范围的 number 变量。

● <c:remove>

<c:remove>主要用来移除变量。

语法

```
<c:remove var="varName" [scope="{ page|request|session|application }"] />
```

属性

名 称	说 明	EL	类型	必须	默认值
var	欲移除的变量名称	N	String	是	无
scope	var 变量的 JSP 范围	N	String	否	page

说明

`<c:remove>`必须要有 `var` 属性，即要被移除的属性名称，`scope` 则可有可无，例如：

```
<c:remove var="number" scope="session" />
```

将 `number` 变量从 `Session` 范围中移除。若我们不设定 `scope`，则 `<c:remove>` 将会从 `Page`、`Request`、`Session` 及 `Application` 中顺序寻找是否存在名称为 `number` 的数据，若能找到时，则将它移除掉，反之则不会做任何的事情。

范例

笔者在这里写一个使用到 `<c:set>` 和 `<c:remove>` 的范例，能让读者可以更快地了解如何使用它们，此范例的名称为 `Core_set_remove.jsp`。

■ `Core_set_remove.jsp`

```
<%@ page contentType="text/html;charset=GB2312" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<html>
<head>
  <title>CH7 - Core_set_remove.jsp</title>
</head>
<body>

  <h2><c:out value="<c:set>和<c:remove> 的用法" /></h2>

  <c:set scope="page" var="number">
    <c:out value="\${1+1}" />
  </c:set>

  <c:set scope="request" var="number">
    <%= 3 %>
  </c:set>

  <c:set scope="session" var="number">
    4
  </c:set>

  初始设置
  <table border="1" width="30%">
  <tr>
    <th>pageScope.number</th>
    <td><c:out value="\${pageScope.number}" default="No Data" /></td>
  </tr>
  <tr>
    <th>requestScope.number</th>
    <td><c:out value="\${requestScope.number}" default="No Data" /></td>
```

JSP2.0 技术手册

```

</tr>
<tr>
  <th>sessionScope.number</th>
  <td><c:out value="${sessionScope.number}" default="No Data" /></td>
</tr>
</table><br>

<c:out value='<c:remove var="number" scope="page" />之后' />
<c:remove var="number" scope="page" />
<table border="1" width="30%">
<tr>
  <th>pageScope.number</th>
  <td><c:out value="${pageScope.number}" default="No Data" /></td>
</tr>
<tr>
  <th>requestScope.number</th>
  <td><c:out value="${requestScope.number}" default="No Data" /></td>
</tr>
<tr>
  <th>sessionScope.number</th>
  <td><c:out value="${sessionScope.number}" default="No Data" /></td>
</tr>
</table><br>

<c:out value='<c:remove var="number" />之后' />
<c:remove var="number" />
<table border="1" width="30%">
<tr>
  <th>pageScope.number</th>
  <td><c:out value="${pageScope.number}" default="No Data" /></td>
</tr>
<tr>
  <th>requestScope.number</th>
  <td><c:out value="${requestScope.number}" default="No Data" /></td>
</tr>
<tr>
  <th>sessionScope.number</th>
  <td><c:out value="${sessionScope.number}" default="No Data" /></td>
</tr>
</table>
</body>
</html>

```

笔者一开始各在 Page、Request 和 Session 三个属性范围中储存名称为 number 的变量。然后先使用 `<c:remove var="number" scope="page" />` 把 Page 中的 number 变量移除，最后再使用 `<c:remove var="number" />` 把所有属性范围中 number 的变量移除。*Core_set_remove.jsp* 的执行结果如图 7-4 所示：

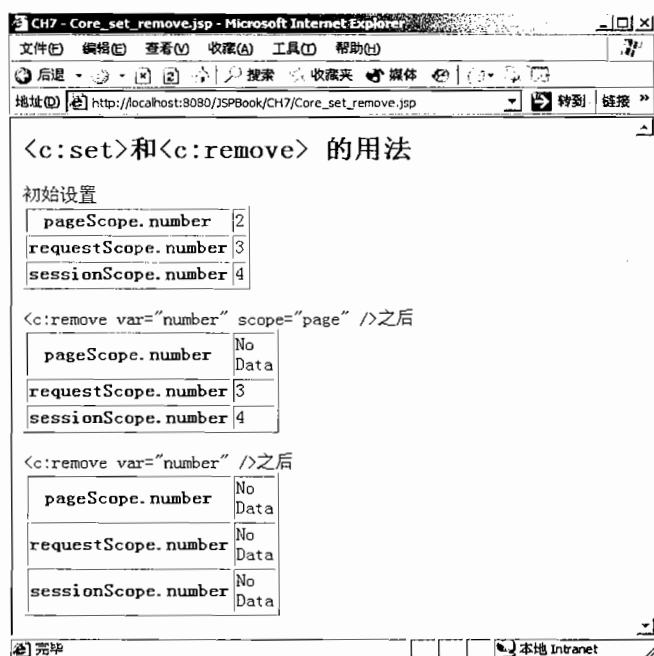


图 7-4 Core_set_remove.jsp 的执行结果

● <c:catch>

<c:catch>主要用来处理产生错误的异常状况，并且将错误信息储存起来。

语法

```
<c:catch [var="varName"] >
... 欲抓取错误的部分
</c:catch>
```

属性

名 称	说 明	EL	类型	必须	默认值
var	用来储存错误信息的变量	N	String	否	无

说明

<c:catch>主要将可能发生错误的部分放在<c:catch>和</c:catch>之间。如果真的发生错误，可以将错误信息储存至 varName 变量中，例如：

```
<c:catch var="message">
: //可能发生错误的部分
</c:catch>
```

JSP2.0 技术手册

另外, 当错误发生在<c:catch>和</c:catch>之间时, 则只有<c:catch>和</c:catch>之间的程序会被中止忽略, 但整个网页不会被中止。

范例

笔者写一个简单的范例, 文件名为 *Core_catch.jsp*, 来让大家看一下<c:catch>的使用方式。

■ Core_catch.jsp

```
<%@ page contentType="text/html; charset=GB2312 " %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<html>
<head>
  <title>CH7 - Core_catch.jsp</title>
</head>
<body>

<h2><c:out value="<c:catch> 的用法" /></h2>

<c:catch var="error_Message">
<%
  String eFormat = "not number";
  int i = Integer.parseInt(eFormat);
%>
</c:catch>
${error_Message}
</body>
</html>
```

笔者将一个字符串转成数字, 如果字符串可以转为整数, 则不会发生错误。但是这里笔者故意传入一个不能转成数字的字符串, 让<c:catch>之间产生错误。当错误发生时, 它会自动将错误存到 error_Message 变量之中, 最后再用<c:out>把错误信息显示出来, 执行结果如图 7-5 所示。

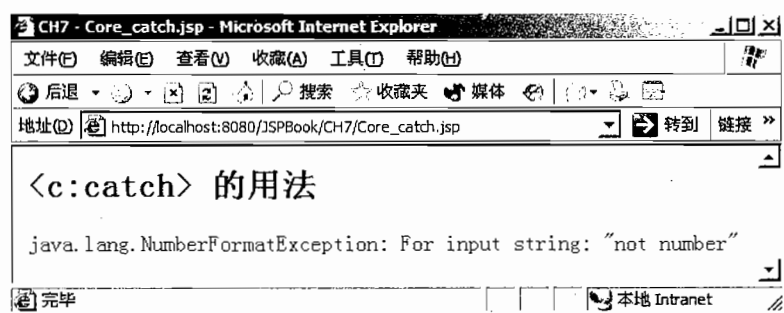


图 7-5 Core_catch.jsp 的执行结果

可以发现到网页确实显示格式错误的信息。如果我们不使用<c:catch>, 而把范例中的<c:catch>和</c:catch>拿掉, 结果如图 7-6 所示。

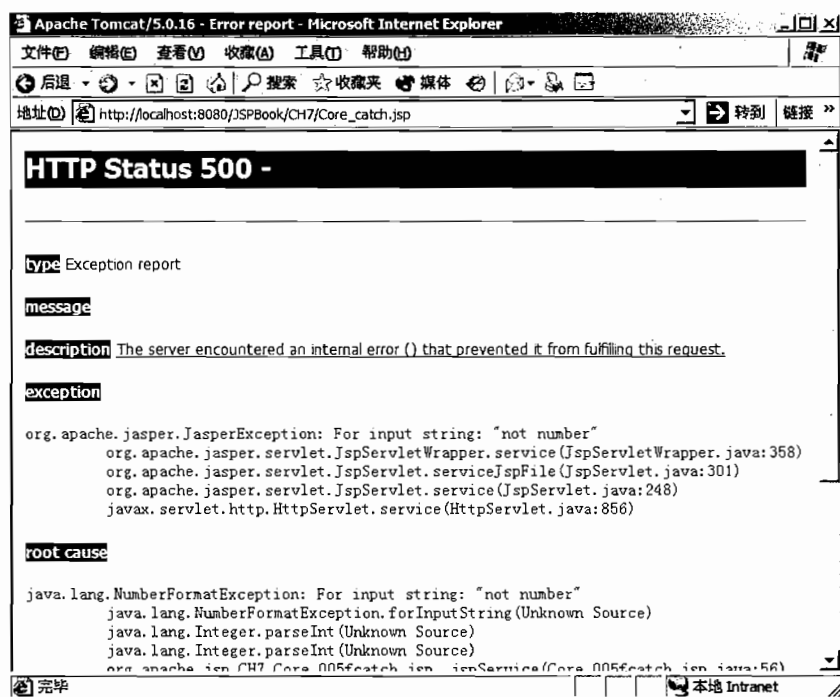


图 7-6 Core_catch.jsp 没有<c:catch>和</c:catch>的执行结果

7-2-2 流程控制

流程控制分类中包含四个标签: <c:if>、<c:choose>、<c:when>和<c:otherwise>, 笔者依此顺序依次说明这四个标签的使用。

● <c:if>

<c:if>的用途就和我们一般在程序中用的 if 一样。

语法

语法1: 没有本体内容(body)

```
<c:if test="testCondition" var="varName"
      [scope="{page|request|session|application}"] />
```

语法2: 有本体内容

```
<c:if test="testCondition" [var="varName"]
      [scope="{page|request|session|application}"]>
```

本体内容

```
</c:if>
```

属性

名称	说明	EL	类型	必须	默认值
test	如果表达式的结果为 true, 则执行本体内容, false 则相反	Y	boolean	是	无
var	用来储存 test 运算后的结果, 即 true 或 false	N	String	否	无
scope	var 变量的 JSP 范围	N	String	否	page

说明

`<c:if>` 标签必须要有 test 属性, 当 test 中的表达式结果为 true 时, 则会执行本体内容; 如果为 false, 则不会执行。例如: `${param.username == 'admin'}`, 如果 param.username 等于 admin 时, 结果为 true; 若它的内容不等于 admin 时, 则为 false。

接下来看下列的范例:

```
<c:if test="${param.username == 'admin'}">
ADMIN 您好!! //body 部分
</c:if>
```

如果名称等于 admin, 则会显示"ADMIN 您好!!"的动作, 如果相反, 则不会执行`<c:if>`的 body 部分, 所以不会显示"ADMIN 您好!! //body 部分"。另外`<c:if>`的本体内容除了能放纯文字, 还可以放任何 JSP 程序代码(Scriptlet)、JSP 标签或者 HTML 码。

除了 test 属性之外, `<c:if>`还有另外两个属性 var 和 scope。当我们执行`<c:if>`的时候, 可以将这次判断后的结果存放到属性 var 里; scope 则是设定 var 的属性范围。哪些情况才会用到 var 和 scope 这两个属性呢? 例如: 当表达式过长时, 我们会希望拆开处理, 或是之后还须使用此结果时, 也可以用它先将结果暂时保留, 以便日后使用。

范例

笔者写了一个简单的范例, 名称为 *Core_if.jsp*。

■ Core_if.jsp

```
<%@ page contentType="text/html; charset=GB2312 " %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>

<html>
<head>
<title>CH7 - Core_if.jsp</title>
</head>
<body>

<h2><c:out value="<c:if> 的用法" /></h2>

<c:if test="${param.username == 'Admin'}" var="condition" scope="page">
您好 Admin 先生
</c:if><br>
```

JSP2.0 技术手册

```

执行结果为: ${condition}
</body>
</html>

```

笔者在判断用户送来的参数时, 如果 username 的值等于 Admin 时, 则会将 condition 设为 true 并存放于 pageScope 中, 否则存放于 condition 中, 最后再显示结果。因为 JSTL 会自动找寻 condition 所存在的属性范围, 因此只须使用 `${condition}`, 而不用 `${pageScope.condition}`。Core_if.jsp 的执行结果如图 7-7。

注意

执行本范例时, 请在 Core_if.jsp 后加上 ?username=Admin。



图 7-7 Core_if.jsp 的执行结果

● <c:choose>

<c:choose>本身只当做 <c:when> 和 <c:otherwise> 的父标签。

语法

```

<c:choose>
    本体内内容( <when> 和 <otherwise> )
</c:choose>

```

属性

无

限制

<c:choose>的本体内内容只能有:

- 空白
- 1 或多个 <c:when>
- 0 或多个 <c:otherwise>

说明

若使用<c:when>和<c:otherwise>来做流程控制时，两者都必须为<c:choose>的子标签，即：

```
<c:choose>
:
  <c:when>
  </c:when>
:
  <c:otherwise>
  </c:otherwise>
:
</c:choose>
```

● <c:when>

<c:when> 的用途就和我们一般在程序中用的 when 一样。

语法

```
<c:when test="testCondition" >
  本内容
</c:when>
```

属性

名 称	说 明	EL	类型	必须	默认值
test	如果表达式的结果为 true，则执行本内容，false 则相反	Y	boolean	是	无

限制

- <c:when> 必须在 <c:choose> 和 </c:choose>之间
- 在同一个 <c:choose> 中时，<c:when> 必须在 <c:otherwise> 之前

说明

<c:when>必须有 test 属性，当 test 中的表达式结果为 true 时，则会执行本内容；如果为 false 时，则不会执行。

● <c:otherwise>

在同一个 <c:choose> 中，当所有 <c:when> 的条件都没有成立时，则执行 <c:otherwise> 的本内容。

语法

```
<c:otherwise>
  本 体 内 容
</c:otherwise>
```

属性

无

限制

- <c:otherwise> 必须在 <c:choose> 和 </c:choose> 之间
- 在同一个 <c:choose> 中时, <c:otherwise> 必须为最后一个标签

说明

在同一个 <c:choose> 中, 假若所有 <c:when> 的 test 属性都不为 true 时, 则执行 <c:otherwise> 的本体内容。

范例

笔者举一个典型的 <c:choose>、<c:when>和<c:otherwise>范例:

```
<c:choose>

  <c:when test="${condition1}">
    condition1 为 true
  </c:when>

  <c:when test="${ condition2}">
    condition2 为 true
  </c:when>

  <c:otherwise>
    condition1 和 condition2 都为 false
  </c:otherwise>

</c:choose>
```

范例说明: 当condition1为true时, 会显示“condition1为true”; 当condition1为false且condition2为true时, 会显示“condition2为true”, 如果两者都为false, 则会显示“condition1和condition2都为false”。

注意

假若 condition1 和 condition2 两者都为 true 时, 此时只会显示"condition1 为 true", 这是因为在同一个<c:choose>下, 当有好几个<c:when>都符合条件时, 只能有一个<c:when>成立。

JSP2.0 技术手册

7-2-3 迭代操作

迭代(Iterate)操作主要包含两个标签: <c:forEach>和<c:forEachTokens>, 笔者依此顺序依次说明这两个标签的使用。

● <c:forEach>

<c:forEach> 为循环控制, 它可以将集合(Collection)中的成员循序浏览一遍。运作方式为当条件符合时, 就会持续重复执行<c:forEach>的本体内容。

语法

语法1: 迭代一集合对象之所有成员

```
<c:forEach [var="varName"] items="collection" [varStatus="varStatusName"]  
          [begin="begin"] [end="end"] [step="step"]>
```

本体内容

```
</c:forEach>
```

语法2: 迭代指定的次数

```
<c:forEach [var="varName"] [varStatus="varStatusName"] begin="begin"  
          end="end" [step="step"]>
```

本体内容

```
</c:forEach>
```

属性

名称	说明	EL	类型	必须	默认值
var	用来存放现在指到的成员	N	String	否	无
items	被迭代的集合对象	Y	Arrays Collection Iterator Enumeration Map String	否	无
varStatus	用来存放现在指到的相关成员信息	N	String	否	无
begin	开始的位置	Y	int	否	0
end	结束的位置	Y	int	否	最后一个成员
step	每次迭代的间隔数	Y	int	否	1

限制

- 假若有 begin 属性时, begin 必须大于等于 0
- 假若有 end 属性时, 必须大于 begin

JSP2.0 技术手册

- 假若有 step 属性时, step 必须大于等于 0

Null 和 错误处理

- 假若 items 为 null 时, 则表示为一空的集合对象
- 假若 begin 大于或等于 items 时, 则迭代不运算

说明

如果要循序浏览一个集合对象, 并将它的内容显示出来, 就必须有 items 属性。

范例

下面的范例 *Core_forEach.jsp* 是将数组中的成员一个个显示出来的:

■ *Core_forEach.jsp*

```
<%@ page contentType="text/html;charset=GB2312 " %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<html>
<head>
  <title>CH7 - Core_forEach.jsp</title>
</head>
<body>

<h2><c:out value="<c:forEach> 的用法" /></h2>

<%
  String atts[] = new String [5];
  atts[0]="hello";
  atts[1]="this";
  atts[2]="is";
  atts[3]="a";
  atts[4]="pen";
  request.setAttribute("atts", atts);
%>

<c:forEach items="${atts}" var="item" >
  ${item}</br>
</c:forEach>

</body>
</html>
```

在上述范例中, 笔者先产生一个字符串数组, 然后将此数组 atts 储存至 Request 的属性范围中, 再用<c:forEach>将它循序浏览一遍。这里 items 表示被浏览的集合对象, var 用来存放指定的集合对象中成员, 最后使用<c:out>将 item 的内容显示出来, 执行结果如图 7-8 所示。



图 7-8 Core_forEach.jsp 的执行结果

注意

varName 的范围只存在<c:forEach>的本体中, 如果超出了本体, 则不能再取得 varName 的值。上个例子中, 若\${item} 是在</c:forEach>之后执行时, 如:

```
<c:forEach items="${atts}" var="item" >
</c:forEach>
${item}</br>
```

\${item}则不会显示 item 的内容。

<c:forEach>除了支持数组之外, 还有标准 J2SE 的集合类型, 例如: ArrayList、List、LinkedList、Vector、Stack 和 Set 等等; 另外还包括 java.util.Map 类的对象, 例如: HashMap、Hashtable、Properties、Provider 和 Attributes。

<c:forEach>还有 begin、end 和 step 这三种属性: begin 主要用来设定在集合对象中开始的位置(注意: 第一个位置为 0); end 用来设定结束的位置; 而 step 则是用来设定现在指到的成员和下一个将被指到成员之间的间隔。我们将之前的范例改成如下:

■ Core_forEach1.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<html>
<head>
<title>CH7 - Core_forEach1.jsp</title>
</head>
<body>

<h2><c:out value="<c:forEach> begin、end 和 step 的用法" /></h2>

<%
```

JSP2.0 技术手册

```

String atts[] = new String [5];
atts[0]="hello";
atts[1]="this";
atts[2]="is";
atts[3]="a";
atts[4]="pen";
request.setAttribute("atts", atts);
%>

<c:forEach items="${atts}" var="item" begin="1" end="4" step="2" >
${item}</br>
</c:forEach>

</body>
</html>

```

<c:forEach>中指定的集合对象 atts 将会从第 2 个成员开始到第 5 个成员, 并且每执行一次循环都会间隔一个成员浏览。因此结果是只显示 atts[1]和 atts[3]的内容, 如图 7-9 所示。

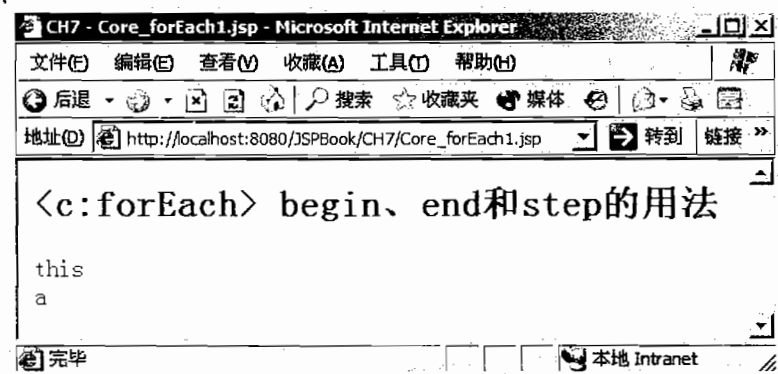


图 7-9 Core_forEach1.jsp 的执行结果

为了方便详细介绍 begin、end 和 step 的不同设定下所产生的结果, 笔者将上面的范例改成如下:

```

<%
int atts[] = {1,2,3,4,5,6,7,8,9,10};
request.setAttribute("atts", atts);
%>
<c:forEach items="${atts}" var="item" begin="0" end="9" step="1" >
${item}</br>
</c:forEach>

```

这里笔者改变 begin、end 和 step 的值时, 在网页上输出结果的变化如表 7-4。

表 7-4

begin	end	step	结 果
-	-	-	1 2 3 4 5 6 7 8 9 10
5	-	-	6 7 8 9 10
-	5	-	1 2 3 4 5 6
-	-	5	1 6
5	5	-	6
5	5	5	6
0	8	2	1 3 5 7 9
0	8	3	1 4 7
0	8	4	1 5 9
15	20	-	无
20	8	-	空白结果
0	20	-	1 2 3 4 5 6 7 8 9 10

从表 7-4 中可以发现：

- (1) 当 begin 超过 end 时将会产生空的结果；
- (2) 当 begin 虽然小于 end 的值，但是当两者都大过容器的大小时，将不会输出任何东西；
- (3) 最后如果只有 end 的值超过集合对象的大小，则输出就和没有设定 end 的情况相同；
- (4) <c:forEach>并不只是用来浏览集合对象而已，读者可以从表 7-4 中发现，items 并不是一定要有的属性，但是当没有使用 items 属性时，就一定要使用 begin 和 end 这两个属性。下面为一个简单的范例：

■ Core_forEach2.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<html>
<head>
  <title>CH7 - Core_forEach2.jsp</title>
</head>
<body>

<h2><c:out value="<c:forEach> 循环" /></h2>

<c:forEach begin="1" end="10" var="item" >
  ${item}</br>
</c:forEach>

</body>
</html>
```

上述范例中，我们并没有执行浏览集合对象，只是设定 begin 和 end 属性的值，这样它就变成一个普通的循环。此范例是将循环设定为：从 1 开始跑到 10，总共会重复循环 10 次，并

且将数字放到 item 的属性当中。`Core_forEach2.jsp` 的执行结果如图 7-10 所示。



图 7-10 `Core_forEach2.jsp` 的执行结果

当然它也可以搭配 `step` 使用，如果将 `step` 设定为 2，结果如图 7-11 所示。



图 7-11 当 `step` 设定为 2 时的结果

另外，`<c:forEach>` 还提供 `varStatus` 属性，主要用来存放现在指到之成员的相关信息。例如：我们写成 `varStatus="s"`，那么将会把信息存放在名称为 `s` 的属性当中。`varStatus` 属性还有另外四个属性：`index`、`count`、`first` 和 `last`，它们各自代表的意义如表 7-5：

表 7-5

属 性	类 型	意 义
<code>index</code>	<code>number</code>	现在指到成员的索引
<code>count</code>	<code>number</code>	总共指到成员的总数
<code>first</code>	<code>boolean</code>	现在指到的成员是否为第一个成员
<code>last</code>	<code>boolean</code>	现在指到的成员是否为最后一个成员

我们可以使用 `varStatus` 属性取得循环正在浏览之成员的信息，下面为一个简单的范例：

■ Core_forEach3.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<html>
<head>
  <title>CH7 - Core_forEach3.jsp</title>
</head>
<body>

<h2><c:out value="<c:forEach> varStatus 的四种属性" /></h2>

<%
  String atts[] = new String [5];
  atts[0]="hello";
  atts[1]="this";
  atts[2]="is";
  atts[3]="a";
  atts[4]="pen";
  request.setAttribute("atts", atts);
%>
<c:forEach items="${atts}" var="item" varStatus="s">
<h2><c:out value="${item}" />的四种属性: </h2>
index: ${s.index}</br>
count: ${s.count}</br>
first: ${s.first}</br>
last: ${s.last}</br>
</c:forEach>

</body>
</html>
```

执行结果如图 7-12 所示。

● <c:forTokens>

`<c:forTokens>` 用来浏览一字符串中所有的成员，其成员是由定义符号(delimiters)所分隔的。

语法

```
<c:forTokens items="stringOfTokens" delims="delimiters" [var="varName"]
  [varStatus="varStatusName"] [begin="begin"] [end="end"] [step="step"]>
  本体内容
</c:forTokens>
```




图 7-12 Core_forEach3.jsp 的执行结果

属性

名 称	说 明	EL	类 型	必 须	默认值
var	用来存放现在指到的成员	N	String	否	无
items	被迭代的字符串	Y	String	是	无
delims	定义用来分割字符串的字符	N	String	是	无
varStatus	用来存放现在指到的相关成员信息	N	String	否	无
begin	开始的位置	Y	int	否	0
end	结束的位置	Y	int	否	最后一个成员
step	每次迭代的间隔数	Y	int	否	1

限制

- 假若有 begin 属性时, begin 必须大于等于 0
- 假若有 end 属性时, 必须大于 begin
- 假若有 step 属性时, step 必须大于等于 0

Null 和 错误处理

- 假若 items 为 null 时, 则表示为--空的集合对象
- 假若 begin 大于或等于 items 的大小时, 则迭代不运算

说明

<c:forTokens>的 begin、end、step、var 和 varStatus 用法都和<c:forEach>一样，因此，笔者在这里就只介绍 items 和 delims 两个属性：items 的内容必须为字符串；而 delims 是用来分割 items 中定义的字符串之字符。

范例

下面为一个典型的<c:forTokens>的范例：

```
<c:forTokens items="A,B,C,D,E" delims="," var="item" >
  ${item}
</c:forTokens>
```

上面范例执行后，将会在网页中输出 ABCDE。它会把符号“,”当做分割的标记，拆成 5 个部分，也就是执行循环 5 次，但是并没有将 A,B,C,D,E 中的“,”显示出来。items 也可以放入 EL 的表达式，如下：

```
<%
  String phoneNumber = "123-456-7899";
  request.setAttribute("userPhone", phoneNumber);
%>
<c:forTokens items="${userPhone}" delims="-" var="item" >
  ${item}
</c:forTokens>
```

这个范例将会在网页上打印 1234567899，也就是把 123-456-7899 以“-”当做分割标记，将字符串拆为 3 份，每执行一次循环就将浏览的部分放到名称为 item 的属性当中。delims 不只指定一种字符来分割字符串，它还可以一次设定多个分割字符串用的字符。如下面这个范例：

```
<c:forTokens items="A,B;C-D,E" delims=";-," var="item" >
  ${item}
</c:forTokens>
```

此范例会在网页输出 ABCDE，也就是说，delims 可以一次设定所有想当做分割字符串用的字符。其实用<c:forEach>也能做到分割字符串，写法如下：

```
<c:forEach items="A,B,C,D,E" var="item" >
  ${item}
</c:forEach>
```

上述范例同样也会在网页输出 ABCDE。<c:forEach>并没有 delims 这个属性，因此<c:forEach>无法设定分割字符串用的字符，而<c:forEach>分割字符串用的字符只有“,”，这和使用<c:forTokens>，delims 属性设为“,”的结果相同。所以如果使用<c:forTokens>来分割字符串，功能和弹性上会比使用<c:forEach>来得较大。

7-2-4 URL 操作

JSTL 包含三个与 URL 操作有关的标签，它们分别为：<c:import>、<c:redirect>和<c:url>。它们主要的功能是：用来将其他文件的内容包含起来、网页的导向，还有 url 的产生。笔者将

JSP2.0 技术手册

依序介绍这三个标签。

● <c:import>

<c:import> 可以把其他静态或动态文件包含至本身 JSP 网页。它和 JSP Action 的 <jsp:include>最大的差别在于:<jsp:include>只能包含和自己同一个 web application 下的文件;而<c:import>除了能包含和自己同一个 web application 的文件外,亦可以包含不同 web application 或者是其他网站的文件。

语法

语法1:

```
<c:import url="url" [context="context"] [var="varName"]
         [scope="{page|request|session|application}"]
         [charEncoding="charEncoding"]>
```

本体内容

```
</c:import>
```

语法2:

```
<c:import url="url" [context="context"] varReader="varReaderName"
         [charEncoding="charEncoding"]>
```

本体内容

```
</c:import>
```

属性

名 称	说 明	EL	类 型	必须	默认值
url	一文件被包含的地址	Y	String	是	无
context	相同 Container 下, 其他 web 站台必须以 "/" 开头	Y	String	否	无
var	储存被包含的文件的内容(以 String 类型存入)	N	String	否	无
scope	var 变量的 JSP 范围	N	String	否	Page
charEncoding	被包含文件之内容的编码格式	Y	String	否	无
varReader	储存被包含的文件的内容(以 Reader 类型存入)	N	String	否	无

Null 和 错误处理

- 假若 url 为 null 或空时, 会产生 JspException

说明

首先<c:import>中必须要有 url 属性, 它是用来设定被包含网页的地址。它可以为绝对地址或是相对地址, 使用绝对地址的写法如下:

```
<c:import url="http://java.sun.com" />
```

`<c:import>`就会把 `http://java.sun.com` 的内容加到网页中。

另外`<c:import>`也支持 FTP 协议, 假设现在有一个 FTP 站台, 地址为 `ftp.javaworld.com.tw`, 它里面有一个文件 `data.txt`, 那么可以写成如下方式将其内容显示出来:

```
<c:import url="ftp://ftp.cse.yzu.edu.tw/data.txt" />
```

如果是使用相对地址, 假设存在一个文件名为 `Hello.jsp`, 它和使用`<c:import>`的网页存在于同一个 `webapps` 的文件夹时, `<c:import>`的写法如下:

```
<c:import url="Hello.jsp" />
```

如果以 `/` 开头, 那么就表示跳到 web 站台的根目录下, 以 Tomcat 为例, 即 `webapps` 目录。假设一个文件为 `hello.txt`, 存在于 `webapps/examples/images` 里, 而 context 为 `examples`, 可以写成以下方式将 `hello.txt` 文件包含进我们的 JSP 页面之中:

```
<c:import url="images/hello.txt" />
```

接下来如果要包含在同一个服务器上, 但并非同一个 web 站台的文件时, 就必须加上 context 属性。假设此服务器上另外还有一个 web 站台, 名为 `others`, `others` 站台底下有一个文件夹为 `jsp`, 且里面有 `index.html` 这个文件, 那么就可以写成如下方式将此文件包含进来:

```
<c:import url="/jsp/index.html" context="/others" />
```

注意

被包含文件的 web 站台必须在 `server.xml` 中定义过, 且`<Context>`的 `crossContext` 属性值必须为 `true`, 这样一来, `others` 目录下的文件才可以被其他 web 站台调用。

`server.xml` 的设置范例如下:

```
<Context path="/others" docBase="others" debug="0"
        reloadable="true" crossContext="true"/>
```

除此之外, `<c:import>`也提供 `var` 和 `scope` 属性。当 `var` 属性存在时, 虽然同样会把其他文件的内容包含进来, 但是它并不会输出至网页上, 而是以 `String` 的类型储存至 `varName` 中。`scope` 则是设定 `varName` 的范围。储存之后的数据, 我们在需要用时, 可以将它取出来, 代码如下:

```
<c:import url="/images/hello.txt" var="s" scope="session" />
```

我们可以把常重复使用的商标、欢迎语句或者是版权声明, 用此方法储存起来, 想输出在网页上时, 再把它导入进来。假若想要改变文件内容时, 可以只改变被包含的文件, 不用修改其他网页。

另外, 可以在`<c:import>`的本体内容中使用`<c:param>`, 它的功用主要是: 可以将参数传

递给被包含的文件，它有两个属性 `name` 和 `value`，如表 7-6 所示：

表 7-6

名 称	说 明	EL	类 型	必 须	默认值
name	参数名称	Y	String	是	无
value	参数的值	Y	String	否	本体内容

这两个属性都可以使用 EL，所以我们写成如下形式：

```
<c:import url="http://java.sun.com" >
<c:param name="test" value="1234" />
</c:import>
```

这样的做法等于是包含一个文件，并且所指定的网址会变成如下：

```
http://java.sun.com?test=1234
```

范例

下面为一用到 `<c:import>`、`<c:param>` 及属性范围的范例，`Core_import.jsp` 和 `Core_imported.jsp`：

■ Core_import.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<html>
<head>
  <title>CH7 - Core_import.jsp</title>
</head>
<body>

<h2><c:out value="<c:import> 的用法" /></h2>

<c:set var="input1" value="使用属性范围传到 Core_imported.jsp 中"
  scope="request" /><hr/>

<c:import url="Core_imported.jsp" charEncoding="GB2312" >
<c:param name="input2" value="使用<c:param>传到 Core_imported.jsp 中" />
</c:import><hr/>

${output1}

</body>
</html>
```

程序中，笔者分别使用 `<c:set>` 和 `<c:param>` 来传递参数。

■ Core_imported.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>

<html>
<head>
<title>CH7 - Core_imported.jsp</title>
</head>
<body>

<fmt:requestEncoding value="GB2312" />
<c:set var="output1" value="使用属性范围传到 Core_import.jsp 中"
scope="request" />
${input1}</br>
<c:out value="${param.input2}" escapeXml="true" />

</body>
</html>
```

Core_imported.jsp 是被包含的文件，它会将从 Core_import.jsp 传来的参数分别输出到页面上，必须注意的是 input1 参数是使用属性范围来传递的，因此可以直接用 \${input1} 来得到参数，而 input2 则必须使用 \${param.input2} 来得到参数。

此外，笔者还使用 <c:set> 来传递值给 Core_import.jsp，这就是 <c:param> 无法做到的动作，<c:param> 只能从包含端抛给被包含端，但是在属性范围中，可以让包含端也能得到被包含端传来的数据。Core_import.jsp 的执行结果如图 7-13 所示：



图 7-13 Core_import.jsp 的执行结果

● <c:url>

<c:url> 主要用来产生一个 URL。

语法

语法1: 没有本体内容

```
<c:url value="value" [context="context"] [var="varName"]
      [scope="{page|request|session|application}"] />
```

语法2: 本体内容代表查询字符串(Query String)参数

```
<c:url value="value" [context="context"] [var="varName"]
      [scope="{page|request|session|application}"] >
  <c:param> 标签
</c:url>
```

属性

名称	说 明	EL	类型	必须	默认值
value	执行的 URL	Y	String	是	无
context	相同 Container 下, 其他 web 站台必须以 "/" 开头	Y	String	否	无
var	储存被包含文件的内容(以 String 类型存入)	N	String	否	无
scope	var 变量的 JSP 范围	N	String	否	Page

说明

在这里笔者直接使用例子来说明。

```
<c:url value="http://www.javaworld.com.tw " >
<c:param name="param" value="value"/>
</c:url>
```

读者可以发现<c:url>也可以搭配<c:param>使用, 上面执行结果将会产生一个网址为 `http://www.javaworld.com.tw?param=value`, 我们更可以搭配 HTML 的<a>使用, 如下:

```
<a href="
  <c:url value="http://www.javaworld.com.tw " >
  <c:param name="param" value="value"/>
</c:url>">台湾 Java 技术论坛</a>
```

另外<c:url>还有三个属性, 分别为 context、var 和 scope。context 属性和之前的<c:import>相同, 可以用来产生一个其他 web 站台的网址。如果<c:url>有 var 属性时, 则网址会被存到 varName 中, 而不会直接输出网址。

哪些状况下才会去使用<c:url>? 例如: 当我们须动态产生网址时, 有可能传递的参数不固定, 或者是需要一个网址能连至同服务器的其他 web 站台之文件, 而且<c:url>更可以将产生的网址储存起来重复使用。另外, 在以前我们必须使用相对地址或是绝对地址去取得需要的图文件或文件, 现在我们可以直接利用<c:url>从 web 站台的角度来设定需要的图文件或文件的地址, 如下:

```
" />
```

JSP2.0 技术手册

如此就会自动产生连到 *image* 文件夹里的 *code.gif* 的地址,不再须耗费精神计算相对地址,并且当网域名称改变时,也不用再改变绝对地址。

● <c:redirect>

<c:redirect>可以将客户端的请求从一个 JSP 网页导向到其他文件。

语法

语法1: 没有本体内容

```
<c:redirect url="url" [context="context"] />
```

语法2: 本体内容代表查询字符串(Query String)参数

```
<c:redirect url="url" [context="context"] >
  <c:param>
</c:redirect >
```

属性

名 称	说 明	EL	类 型	必须	默认值
url	导向的目标地址	Y	String	是	无
context	相同 Container 下, 其他 web 站台必须以 “/” 开头	Y	String	否	无

说明

url 就是设定要被导向到的目标地址, 它可以是相对或绝对地址。例如: 我们写成如下:

```
<c:redirect url="http://www.javaworld.com.tw" />
```

那么网页将会自动导向到 <http://www.javaworld.com.tw>。另外, 我们也可以加上 context 这个属性, 用来导向至其他 web 站台上的文件, 例如: 导向到 */others* 下的 */jsp/index.html* 时, 写法如下:

```
<c:redirect url="/jsp/index.html" context="/others" />
```

<c:redirect> 的功能不止可以导向网页, 同样它还可以传递参数给目标文件。在这里我们同样使用<c:param>来设定参数名称和内容。

范例

■ Core_redirect.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html>
```

JSP2.0 技术手册

```

<head>
  <title>CH7 - Core_redirect.jsp</title>
</head>
<body>

<h2><c:out value="<c:redirect> 的用法" /></h2>

<c:redirect url="http://java.sun.com">
<c:param name="param" value="value"/>
</c:redirect>

<c:out value="不会执行喔!!!" />

</body>
</html>

```

执行结果如图 7-14，可以发现网址部分为 `http://java.sun.com/?param=value`：

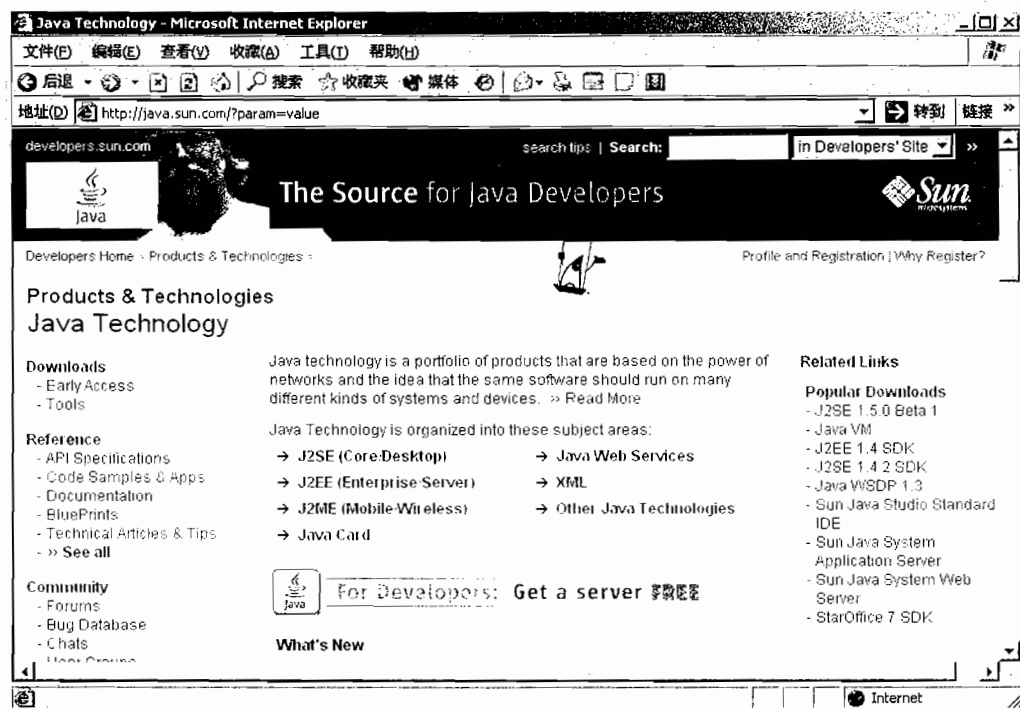


图 7-14 Core_redirect.jsp 的执行结果

7-3 I18N 格式标签库 (I18N-capable formatting tags library)

JSTL 中最重要的功能就是国际化格式(I18N)的支持，此功能可以对一个特定的语言请求做出适合的响应，例如：台湾的用户发出请求时，响应繁体中文的结果。它使用 J2SE 的 ResourceBundle，它会根据不同的地区选择适合的 ResourceBundle。

<fmt:setLocale>用来设置地区，比如<fmt:setLocale value="zh_TW"/>，这等于设定了语言和国家代码为台湾地区，当然还可以指定 ResourceBundle，比如：<fmt:bundle basename="LocalStrings"/>。另外，使用<fmt:requestEncoding>来设定请求的字符编码；<fmt:formatNumber>或<fmt:parseNumber>可以按当地的格式显示数值、货币金额、百分比。

例如：有一个数字 356987.589，在法国，数值格式会变为 356 987,589；在德国，数值格式会变为 356.987,589；而在美国，数值格式会变为 356,987.589。由此可知，同一个数字，在不同的国家中，表示的方式会有所不同。

I18N 格式标签库(I18N)主要包含：国际化、消息和数字日期格式化。详细分类如表 7-7 所示，接下来笔者将为读者一一介绍每个标签的功能。

表 7-7

分 类	功能分类	标签名称
I18N	国际化	setLocale requestEncoding
	消息	bundle message param setBundle
	数字日期格式化	formatNumber formatDate parseDate parseNumber setTimeZone timeZone

JSP 要使用 I18N 格式标签库时，必须使用<%@ taglib %>指令，并且设定 I18N 格式标签库：
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>

7-3-1 国际化

国际化这个分类中包含两个标签，一个是设定语言地区的<fmt:setLocale>；另一个则是设

定请求的字符串编码的<fmt:requestEncoding>。接下来介绍这两个标签：

● <fmt:setLocale>

<fmt:setLocale>用来设定用户的语言地区。

语法

```
<fmt:setLocale value="locale" [variant="variant"]
               [scope="{page|request|session|application}"] />
```

属性

名 称	说 明	EL	类型	必须	默认值
value	地区代码。其中最少要有两个字母的语言代码，如：zh、en，然后也可以再加上两个字母的国家和地区代码，如：US、TW，两者可以由“-”或“_”相连接起来，例如：zh_TW	Y	String / java.util.Locale	是	无
variant	供货商或浏览器的规格。例如：WIN 代表 Windows，Mac 代表 Macintosh	Y	String	否	无
scope	地区设定的套用范围	N	String	否	Page

Null 和 错误处理

- 假若 value 为 null 时，则使用执行中的默认地域(locale)

说明

<fmt:setLocale>一定要有 value 属性，它主要用来设定地区代码。在同一网页中，只要可以套用到地区属性的数据，就会依据现在的设定把数据转换成当地的表示方式。

地区代码设定方面我们可以从 <http://www.w3.org/WAI/ER/IG/ert/iso639.htm> 找到语言代码，在 <http://www.unicode.org/onlinedat/countries.html> 可以找到国家与地区代码，<fmt:setLocal>至少必须设定语言代码。通常我们可以用“_”或“-”来汇编语言和国家与地区代码（见表 7-8）：

表 7-8

区域代码	区域、语言
en	英文
en_US	英文（美国）
zh_TW	中文（台湾）
zh_CN	中文（中国）

只要每设定一次<fmt:setLocal>, 之后的地区设定都会依据最后设定的地区代码。

范例

底下为一个简单的范例, 产生各种地区设定的日期显示方式:

■ Fmt_setLocale.jsp

```
<%@ page import = "java.util.Date" %>
<%@ page contentType="text/html; charset=GB2312" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>

<html>
<head>
  <title>CH7 - Fmt_setLocale.jsp</title>
</head>
<body>

<h2>&lt;fmt:setLocale&gt;</h2>

<%
  Date now = new Date();
  request.setAttribute("d", now);
%>
</br>

JAPANESE(JAPAN): <fmt:setLocale value="ja_JP"/><fmt:formatDate
value= "${d}" /></br>
SPANISH: <fmt:setLocale value="es"/><fmt:formatDate value="${d}" /></br>
FRENCH: <fmt:setLocale value="fr"/><fmt:formatDate value="${d}" /></br>
FRENCH(CANADA): <fmt:setLocale value="fr_CA"/><fmt:formatDate
value= "${d}" /></br>
CHINESE(TAIWAN): <fmt:setLocale value="zh_TW"/><fmt:formatDate
value= "${d}" /></br>
CHINESE(CHINA): <fmt:setLocale value="zh_CN"/><fmt:formatDate
value= "${d}" /></br>

</body>
</html>
```

*Fmt_setLocale.jsp*中, 笔者依序设定: 日本、西班牙、法文、加拿大区法文、台湾地区和
中国, 而<fmt:formatDate value="\${d}"/>用来把日期的显示方式转化成适当区域的表示方法。
*Fmt_setLocale.jsp*的执行结果如图7-15所示。

● <fmt:requestEncoding>

<fmt:requestEncoding>用来设定字符串编码, 功能和 `ServletRequest.setCharacterEncoding()`
相同。

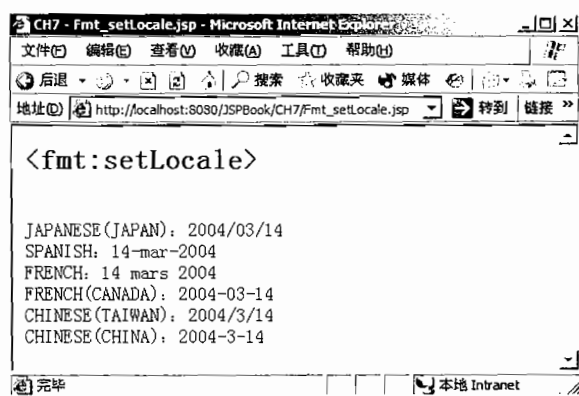


图 7-15 Fmt_setLocale.jsp 的执行结果

语法

```
<fmt:requestEncoding [value="charseName"] />
```

属性

名称	说明	EL	类型	必须	默认值
value	字符串编码	Y	String	否	自动寻找

说明

当我们设定 value 为 GB2312 时, 表示将所有传送过来的字符串皆用 GB2312 编码处理, 此方法可以让我们的 JSP 轻松地处理中文问题。如果没有设定 value 属性, 则它将会自动去寻找合适的编码方式。

7-3-2 消息

消息类中包含四个标签: <fmt:message>、<fmt:param>、<fmt:bundle>和<fmt:setBundle>。它们的主要功用为: 抓取系统设定之语系资源, 让我们可以轻易地做到多国化信息。在使用这些有关国际化信息的标签前, 我们必须知道如何产生存放国际化信息的数据来源。

首先用文字编辑器产生一个文件, 扩展名必须为 properties, 然后文件的内容必须是依照 key = value 的格式, 也就是一个索引对应到一个值。最后将文件放至与 JSP 网页同一个站台的 WEB-INF/classes 目录下。笔者在这里产生一个 MyResource.properties 文件, 内容如下:

■ MyResource.properties

```
Str1=Hello
Str2=My name is Bunny
```

接下来就是介绍如何使用<fmt:message>、<fmt:param>、<fmt:bundle>和<fmt:setBundle>

来取得消息。

- **<fmt:message>**
 <fmt:message>会从指定的资源中把特定关键字中的值抓取出来。

语法

语法1

```
<fmt:message key="messageKey" [bundle="resourceBundle"]  
                    [var="varName"] [scope="{page|request|session|application}"] />
```

语法2

```
<fmt:message key="messageKey" [bundle="resourceBundle"]  
                    [var="varName"] [scope="{page|request|session|application}"]>  
<fmt:param>  
</fmt:message>
```

语法3

```
<fmt:message [bundle="resourceBundle"] [var="varName"]  
                    [scope="{page|request|session|application}"]>
```

索引

```
(<fmt:param> 标签)  
</fmt:message>
```

属性

名 称	说 明	EL	类 型	必 须	默认值
key	索引	Y	String	否	无
bundle	使用的数据来源	Y	LocalizationContext	否	无
var	用来储存国际化信息	N	String	否	无
scope	var 变量的 JSP 范围	N	String	否	Page

Null 和 错误处理

- 假若 key 为 null 或空的时，在网页上会产生 “?????” 的错误信息
- 假若找不到资源文件(.properties)时，在网页上会产生 “???<key>???” 的错误信息

说明

如果<fmt:message>没有 key 属性，那么<fmt:message>将会自动从本体内容中寻找关键字，再从关键字中寻找对应的结果，显示在网页中。bundle 属性则是指定使用的资源对象。var 和 scope 则用来储存要显示的信息。如果有 var 属性时，<fmt:message>将不会把结果显示在网页中，而是

将结果储存在 `varName` 中，所以想要将 `key` 显示在网页时，就必须使用 `<c:out>` 来完成。

假设笔者使用的数据来源为前面所设定的 `MyResource.properties`，因为其中有 `key` 为 `Str1`；`value` 为 `hello`，所以我们可以使用如下方式将其内容显示在网页中：

```
<fmt:message key="str1" />
```

● `<fmt:param>`

当我们从数据来源中抓出内容时，有时要动态设定参数，例如日期或者是用户名称等等，此时就必须使用 `<fmt:param>`。

语法

语法1：通过 `value` 属性设定参数值

```
<fmt:param value="messageParameter" />
```

语法2：通过本体内容设定参数值

```
<fmt:param>
```

本体内容

```
</fmt:param>
```

属性

名 称	说 明	EL	类 型	必 须	默认值
value	增加的变量	Y	Object	否	无

说明

`value` 就是设定要给予的参数，如果没有 `value` 属性，那么就会默认去抓取本体内容当做要给予的参数。

范例

例如：我们在 `MyResource.properties` 文件中，新增一个索引和值如下：

```
Str3=today is {0,date}
```

其中 `{0, date}` 表示为一个动态变量，`0` 代表第一个动态变量；`date` 代表该动态变量的类型。我们再将 `Str3` 改为如下：

```
Str3= Hi ! {0} </br> today is {1, date, long} </br> time is {2, time, full} </br> number is {3, number, #.##}
```

这里笔者定义四个动态变量，然后我们再来使用 `<fmt:param>` 给这四个变量设定一个值：

■ *Fmt_param.jsp*

```
<%@ page import = "java.util.Date" %>
<%@ page import = "java.lang.Double" %>
<%@ page contentType="text/html; charset=GB2312" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>

<html>
<head>
  <title>CH7 - Fmt_param.jsp</title>
</head>
<body>

<h2>&lt;fmt:param&gt; </h2>

<%
  request.setAttribute("now", new Date());
  request.setAttribute("num", new Double(123.45) );
%>
</br>
<fmt:bundle basename="MyResource">
  <fmt:message key="Str3">
    <fmt:param value="JavaWorld in Taiwan" />
    <fmt:param value="{now}" />
    <fmt:param value="{now}" />
    <fmt:param value="{num}" />
  </fmt:message>
</fmt:bundle>
</body>
</html>
```

Fmt_param.jsp 程序中最主要的一段如下:

```
<fmt:bundle basename="MyResource">
  <fmt:message key="Str3">
    <fmt:param value="JavaWorld in Taiwan" />
    <fmt:param value="{now}" />
    <fmt:param value="{now}" />
    <fmt:param value="{num}" />
  </fmt:message>
</fmt:bundle>
```

因为 Str3 有四个变量,所以需要四个<fmt:param>来设定它们,图 7-16 是 *Fmt_param.jsp* 的执行结果。

注意

假若你要显示中文字时,在资源文件(.properties)中,中文字的部分必须为 Unicode,否则无法正确显示。

● **<fmt:bundle>**

<fmt:bundle>主要用来设定本体内容的数据来源。



图 7-16 Fmt_param.jsp 的执行结果

语法

```
<fmt:bundle basename="basename" [prefix="prefix"]>
```

本体内容

```
</fmt:bundle>
```

属性

名称	说 明	EL	类型	必须	默认值
basename	要使用的资源名称, 例如: MyResource	Y	String	是	无
prefix	前置关键字	Y	String	否	无

Null 和 错误处理

- 假若 basename 为 null、空的或找不到资源文件(.properties)时, 在网页上会产生 “???<key>???” 的错误信息。

说明

basename 属性是设定要使用的数据来源, 如果我们的 properties 文件为 MyResource.properties, 那么 basename 的值为 MyResource。

注意

basename 的值千万不可有任何文件类型(即扩展名, 如: .class or .properties), 所以不可用 MyResource.properties。

prefix 用来设定前置关键字, 例如: 当 properties 文件内容如下时:

```
requestinfo.label.method=Method:
requestinfo.label.requesturi=Request URI:
requestinfo.label.protocol=Protocol:
requestinfo.label.pathinfo=Path Info:
requestinfo.label.remoteaddr=Remote Address:
```

那么我们就可以写成如下方式将其内容显示出来:

```
<fmt:bundle basename="MyResource" prefix="requestinfo.label.">
<fmt:message key="protocol"/>
<fmt:message key="pathinfo"/>
:
</fmt:bundle>
```

也就是说, 如果设定好 `prefix`, 就可以无须每次使用 `<fmt:message>` 时都要打上一长串的索引。

● `<fmt:setBundle>`

`<fmt:setBundle>` 可以用来设定默认的数据来源, 或者也可以将它设定到属性范围中。

语法

```
<fmt:setBundle basename="basename" [var="varName"]
[scope="{page|request|session|application}"] />
```

属性

名 称	说 明	EL	类 型	必 须	默认值
basename	要使用的资源名称, 例如: MyResource	Y	String	是	无
var	储存资源的名称	N	String	否	无
scope	var 变量的 JSP 范围	N	String	否	Page

Null 和 错误处理

- 假若 `basename` 为 `null`、空的或找不到资源文件(.properties)时, 在网页上会产生 “`???<key>???`” 的错误信息。

说明

`basename` 设定要使用的数据来源, 和 `<fmt:bundle>` 用法相同。如果没有设定 `var` 时, 那么设定好的数据来源将会变成默认的数据来源, 让在同一网页或同一属性范围中的 `<fmt:message>` 都可以直接默认使用此数据来源。相反, 如果设定 `var` 时, 那么将会把此数据来源存入 `varName` 中, 当 `<fmt:message>` 要使用时, 必须使用 `bundle` 这个属性来指定。如下:

```
<fmt:setBundle basename="MyResource" scope="session" var="myResource"/>
<fmt:message key="str1" bundle="${myResource}"/>
```


如果没有设定 var 时, 则只须写成如下:

```
<fmt:setBundle basename="MyResource" scope="session" />
<fmt:message key="str1" />
```

一般而言, <fmt:bundle>和<fmt:setBundle>都可以搭配<fmt:setLocale>使用。当我们有多种语言的数据来源时, 可以将文件名取成 *MyResource_zh_TW.properties*、*MyResource_en.properties* 和 *MyResource.properties*。当我们将区域设定为 zh_TW, 那么使用<fmt:bundle>或<fmt:setBundle>时, 将会默认读取 *MyResource_zh_TW.properties* 资源文件; 如果<fmt:setLocale>设定为 en 时, 那么将会默认抓取 *MyResource_en.properties* 来使用; 最后如果设定的区域没有符合的文件名, 那么将会使用 *MyResource.properties* 来当做数据来源。代码如下:

```
<fmt:setLocale value="zh_TW" />
<fmt:setBundle basename="MyResource" scope="session" />
<fmt:message key="str1" />
```

如果存在 *MyResource_zh_TW.properties*, 就会先去抓取此文件当做数据来源; 反之, 如果没有 *MyResource_zh_TW.properties*, 则会抓取 *MyResource.properties* 来当做数据来源。

接下来, 笔者将实现一个 I18N 功能的范例程序, 它会先判断浏览器的语系, 然后显示该语系的文字。本范例主要包含三个部分:

- Fmt_i18n.jsp
- Resource_en_US.properties
- Resource_zh_TW.properties

Fmt_i18n.jsp 为主要的程序部分, 它会根据浏览器的语系自动存取该语系的资源文件。笔者在这里建立两种语系的资源文件: 一种是英文的 *Resource_en_US.properties*; 另一种是中文的 *Resource_zh_TW.properties*。

笔者使用 IE 6.0 作为测试用的浏览器, 并且一开始将 IE 的语系设定为中文(台湾), 如图 7-17。

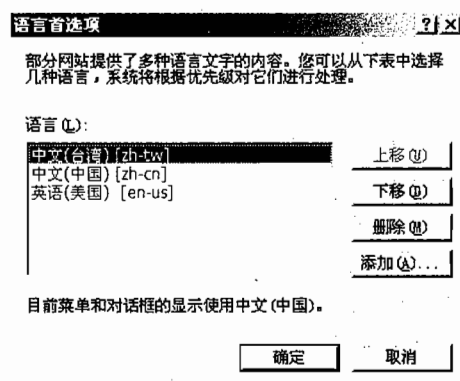


图 7-17 将 IE 的语系设定为中文(台湾)

设定成功之后，我们来执行 *Fmt_i18n.jsp*，结果如图 7-18。



图 7-18 IE 语系为中文(台湾)时，执行 *Fmt_i18n.jsp*

假若今天 IE 语系为英文(美国)时，如图 7-19。

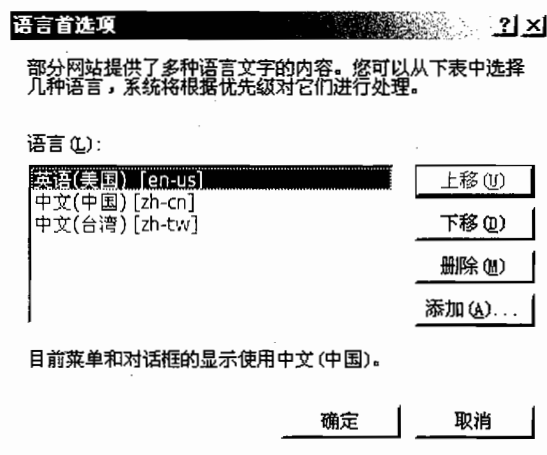


图 7-19 IE 语系设定为英文(美国)

同样地，我们再来执行 *Fmt_i18n.jsp*，你会发现文字、时间和数字的显示方法都不太一样，如图 7-20。

假若今天我们要做一个多语系的站台时，就可以使用这个方式来达成，站台程序部分依旧为同一个程序，不同的只是资源文件，一种语系，一个资源文件。接下来笔者就说明这个范例程序。

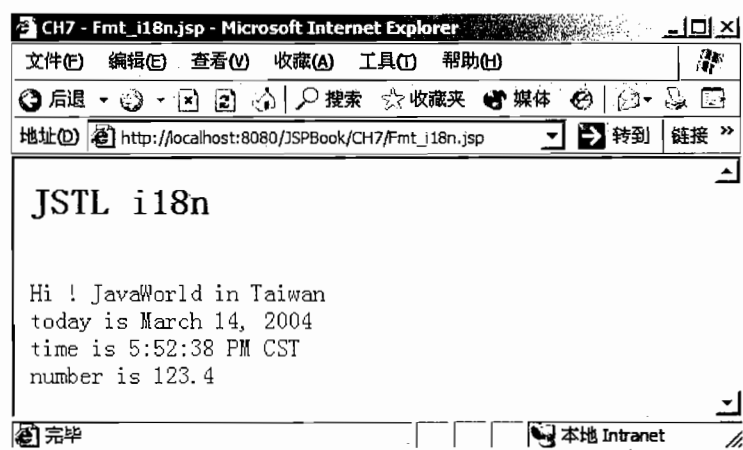


图 7-20 IE 语系为英文(美国)时, 执行 Fmt_i18n.jsp

■ Fmt_i18n.jsp

```
<%@ page import = "java.util.Date" %>
<%@ page import = "java.lang.Double" %>
<%@ page contentType="text/html; charset=GB2312" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>

<html>
<head>
  <title>CH7 - Fmt_i18n.jsp</title>
</head>
<body>

<h2>JSTL i18n </h2>
<%
    request.setAttribute("now", new Date());
    request.setAttribute("num", new Double(123.45) );
%>
</br>
<fmt:bundle basename="Resource">
  <fmt:message key="Str">
    <fmt:param value="JavaWorld in Taiwan" />
    <fmt:param value="{now}" />
    <fmt:param value="{now}" />
    <fmt:param value="{num}" />
  </fmt:message>
</fmt:bundle>
</body>
</html>
```

Fmt_i18n.jsp 和之前 *Fmt_param.jsp* 程序很像, 只不过 *Fmt_i18n.jsp* 是读取 *Resource.properties* 资源文件。接下来我们再来看英文(美国)语系的资源文件 *Resource_en_US.properties*。

■ *Resource_en_US.properties*

```
Str=Hi ! {0} </br> today is {1, date, long} </br> time is {2, time, full}
</br> number is {3, number, #.#}
```

Resource_en_US.properties 有四个动态变量: 第一个是字符串类型; 第二个是 date 类型; 第三个是 time 类型; 最后一个为 number 类型。

注意

{1, date, long} 的 long 表示 date 的显示格式。除了 long 之外, date 的显示格式还有: short、medium、full 等。同理 {2, time, full} 和 {3, number, #.#} 也是相类似的, 这部分, 有兴趣的读者可以自行去查阅 Java J2SDK 1.4.2 API 中的 java.text.MessageFormat。

■ *Resource_zh_TW.properties*

```
Str=\u55E8 ! {0} </br> \u4ECA\u5929\u65E5\u671F\u70BA\uFF1A{1, date, long}
</br> \u6642\u9593\u70BA\uFF1A{2, time, full} </br>
\u6578\u5B57\u70BA\uFF1A{3, number, #.#}
```

Resource_zh_TW.properties 是中文(台湾)语系的资源文件, 它的内容大部分都是一些 Unicode 的符号, 它们都是代表一个中文字。假若你在 *Resource_zh_TW.properties* 中直接输入储存中文, 在执行 *Fmt_i18n.jsp* 时, 就会出现一堆乱码, 无法正确显示中文。

那我们又如何知道中文字的 Unicode 呢? Sun J2SDK 提供了一个好用的转换工具, 名为 native2ascii。这里举个例子, 我在 C:\ 目录下建立一个 *Resource.properties* 文件, 内容如下:

```
Str=中文
```

然后, 在 DOS 窗口下输入指令:

```
C:\>native2ascii -encoding GB2312 Resource.properties
Resource_zh_TW.properties
```

然后它就会自动产生一个名为 *Resource_zh_TW.properties* 资源文件, 内容如下:

```
Str=\u4e2d\u6587
```

其中, "中" 的 Unicode 为 \u4e2d; "文" 的 Unicode 为 \u6587。这样子就能直接把中文转换为 Unicode 了。

7-3-3 数字、日期格式化

数字、日期格式化包含六个标签, 分别为: <fmt:formatNumber>、<fmt:parseNumber>、<fmt:formatDate>、<fmt:parseDate>、<fmt:setTimeZone>和<fmt:timeZone>, 它们皆是用来将数字或日期转换成该区域的格式。

● <fmt:formatNumber>

<fmt:formatNumber> 会依据设定的区域将数字改为适当的格式, 例如: 若有一个数字 5000000.5, 若区地设为台湾时, 在网页中会显示 5,000,000.5。

语法

语法1: 没有本体内容

```
<fmt:formatNumber value="numericValue" [type="{number|currency|percent}"]
    [pattern="customPattern"]
    [currencyCode="currencyCode"]
    [currencySymbol="currencySymbol"]
    [groupingUsed="{true|false}"]
    [maxIntegerDigits="maxIntegerDigits"]
    [minIntegerDigits="minIntegerDigits"]
    [maxFractionDigits="maxFractionDigits"]
    [minFractionDigits="minFractionDigits"]
    [var="varName"]
    [scope="{page|request|session|application}"] />
```

语法2: 本体内容为欲格式化的数字

```
<fmt:formatNumber [type="{number|currency|percent}"]
    [pattern="customPattern"]
    [currencyCode="currencyCode"]
    [currencySymbol="currencySymbol"]
    [groupingUsed="{true|false}"]
    [maxIntegerDigits="maxIntegerDigits"]
    [minIntegerDigits="minIntegerDigits"]
    [maxFractionDigits="maxFractionDigits"]
    [minFractionDigits="minFractionDigits"]
    [var="varName"]
    [scope="{page|request|session|application}"] >
```

欲格式化的数字

● </fmt:formatNumber>

属性

名 称	说 明	EL	类型	必须	默认值
value	欲格式化的数字	Y	String / Number	否	无
type	指定单位(数字、当地货币、百分比)	Y	String	否	number
pattern	格式化数字的样式	Y	String	否	无
currencyCode	ISO-4217 码	Y	String	否	无
currencySymbol	货币符号, 如: \$	Y	String	否	无
groupingUsed	是否区隔数字, 如: 9,123,567	Y	boolean	否	true
maxIntegerDigits	整数部分最多显示多少位数	Y	int	否	无
minIntegerDigits	整数部分最少显示多少位数	Y	int	否	无
maxFractionDigits	小数点后最多显示多少位数	Y	int	否	无
minFractionDigits	小数点后最少显示多少位数	Y	int	否	无
var	储存已格式化之数字	N	String	否	无
scope	var 变量的 JSP 范围	N	String	否	Page

JSP2.0 技术手册

限制

- `currencyCode` 必须符合 ISO 4217 的货币码

Null 和 错误处理

- 假若无法确定区域时，则输出的格式为 `Number.toString()`
- 假若 `pattern` 为 `null` 或空的时，则 `pattern` 将被忽略

说明

`<fmt:formatNumber>` 会依据设定的区域格式化数字的显示方式。例如：5000000.01 在法国是 5 000 000,01；在德国是 5.000.000,01；在台湾地区则是 5,000,000.5。

`type` 则是设定数字所代表的单位，总共有下列三种（见表 7-9）：

表 7-9

type 的类型	功 能	范例 (当 value 为 .5 时)
number	标准数字	0.5
currency	当地货币码	NT\$0.50
percent	百分比	50%

`currencyCode` 为货币代码，例如：美金就是 USD、台币为 TWD。详细的货币代码可以至 http://www.bsi-global.com/Technical+Information/Publications/_Publications/tig90x.doc 查询。`currencySymbol` 则为货币符号，例如：美金是 "\$"。

`groupingUsed` 用来设定是否将数字区隔，例如：100000 区隔后会变成 100,000。所以当 `groupingUsed` 为 `true` 时，数字将会区隔；反之，如果为 `false`，则直接显示数字，不做区隔的动作。

`maxIntegerDigits` 和 `minIntegerDigits` 用来设定数字的整数部分可以表示的位数。如果 `maxIntegerDigits` 小于 `value` 的位数时，那么将会把多的位数部分删掉，例如：当 `value` 为 1234，而 `maxIntegerDigits` 为 3，那么结果将会是 234。如果 `minIntegerDigits` 大于 `value` 的位数时，那么将会用 0 把不足的部分补齐，例如：当 `value` 为 1234，而 `minIntegerDigits` 为 5，那么结果将会是 01234。

`maxFractionDigits` 和 `minFractionDigits` 用来设定数字的小数部分可以表示的位数。如果 `maxFractionDigits` 小于 `value` 的位数时，那么将会把多的部分删掉，例如：当 `value` 为 0.1234，而 `maxFractionDigits` 为 3，那么结果将会是 0.123。如果 `minFractionDigits` 大于 `value` 的位数时，那么将会用 0 把不足的部分补齐。例如：当 `value` 为 0.1234，而 `minFractionDigits` 为 5，那么结果将会是 0.12340。

我们有时候可能会用到科学记号，例如：1.25E40 代表 1.25×10 的 40 次方，`<fmt:formatNumber>` 也有提供产生科学记号的功能。当我们想产生科学记号时必须使用 `pattern` 这个属性，使用方式如下：

```
<fmt:formatNumber value="20512" pattern="###.###E0" />
```

那么结果将会是：

JSP2.0 技术手册

20.512E3

详细的 pattern 设定可以参考 <http://java.sun.com/j2se/1.4.2/docs/api/java/text/DecimalFormat.html>。

最后 var 和 scope 则是用来储存格式化后数字的属性名称和属性范围的。

● <fmt:parseNumber>

<fmt:parseNumber>可以将字符串类型的数字、货币或百分比，都转为数字类型。

语法

语法1: 没有本体内容

```
<fmt:parseNumber value="numericValue" [type="{number|currency|percent}"]
    [pattern="customPattern"]
    [parseLocale=" parseLocale"]
    [integerOnly="{true|false}"]
    [var="varName"]
    [scope="{page|request|session|application}"] />
```

语法2: 本体内容为欲格式化的数字

```
<fmt:parseNumber [type="{number|currency|percent}"]
    [pattern="customPattern"]
    [parseLocale=" parseLocale"]
    [integerOnly="{true|false}"]
    [var="varName"]
    [scope="{page|request|session|application}"] />
```

欲转换为数字的字符串

```
</fmt:formatNumber>
```

属性

名 称	说 明	EL	类型	必须	默认值
value	欲格式化的数字	Y	String / Number	否	无
type	指定单位(数字、当地货币、百分比)	Y	String	否	number
pattern	格式化数字的样式	Y	String	否	无
parseLocale	用来取代默认的地区设定	Y	String / java.util.Locale	否	无
integerOnly	是否只显示整数部分	Y	boolean	否	false
var	储存已格式化之数字	N	String	否	无
scope	var 变量的 JSP 范围	N	String	否	Page

Null 和 错误处理

- 假若 pattern 为 null 或空的时，则 pattern 将被忽略

JSP2.0 技术手册

说明

value 为欲被转换的字符串，下面为一个简单的范例：

```
<fmt:parseNumber value=" 500,800" />
```

结果为显示出 500800 这个数字。type 则是用来设定解析的字符串的类型，例如：当字符串为 NT\$5000 时，我们就必须将 type 设定为 currency，如下：

```
<fmt:parseNumber value="NT$5000" type="currency" />
```

那么在画面中，显示的结果为 5000。

integerOnly 则是设定转换成数字后，是否保留小数点后的数字。如果为 true，则不会保留小数点以后的数字；如果为 false，则会保留小数点之后的数字。

例如：输入的字符串为 25%，那么我们可以使用 pattern 将字符串转换成数字。如下：

```
<fmt:parseNumber value="25%" pattern="00%" />
```

那么将会把 25% 转换成 0.25，并显示在画面中。详细的 pattern 设定可以参考 <http://java.sun.com/j2se/1.4.2/docs/api/java/text/DecimalFormat.html>。

parseLocale 用来设定地区，例如：从数据库抓出数据时，有可能是某地区的各种商品的价格，但是价格通常会依地区不同而有不同的单位，因此可以使用 parseLocale 简化处理字符串。

● <fmt:formatDate>

<fmt:formatDate>主要用来格式化日期、时间。

语法

```
<fmt:formatDate value="date" [type="{time|date|both}"]
[pattern="customPattern"]
    [dateStyle="{default|short|medium|long|full}"]
    [timeStyle="{default|short|medium|long|full}"]
    [timeZone="timeZone"]
    [var="varName"]
    [scope="{page|request|session|application}"] />
```

属性

名 称	说 明	EL	类 型	必 须	默认值
value	欲被格式化的日期时间	Y	java.util.Date	否	无
type	显示的部分(日期、时间或两者)	Y	String	否	date
dateStyle	设定日期部分的显示方式	Y	String	否	default
timeStyle	设定时间部分的显示方式	Y	String	否	default

续表

名 称	说 明	EL	类 型	必 须	默认值
pattern	格式化日期时间的样式	Y	String	否	无
timeZone	设定使用的时区	Y	String / java.util.TimeZone	否	无
var	储存已格式化之日期时间	N	String	否	无
scope	var 变量的 JSP 范围	N	String	否	Page

说明

当我们在网页中写成如下形式：

```
<jsp:useBean id="now" class="java.util.Date"/>
<fmt:formatDate value="${now}" />
```

那么将会在网页中显示 2003/2/17，如果用户的地区为美国时，则显示为 Feb 17, 2003。value 为欲格式化的日期时间。type 设定要显示的部分，如果为 date，则会显示 2003/2/17；如果是 time，则会显示下午 08:13:38；若为 both 时，则会显示 2003/2/17 下午 08:13:38。

dateStyle 和 timeStyle 则是设定日期和时间的表示程度。笔者所在的区域为台湾，设定方式和结果如表 7-10：

表 7-10

设定值	Date 结果	Time 结果
default	2003/2/17	下午 08:24:06
short	2003/2/17	下午 8:23
medium	2003/2/17	下午 08:23:53
long	2003 年 2 月 17 日	下午 08 时 23 分 29 秒
full	2003 年 2 月 17 日 星期一	下午 08 时 23 分 17 秒 CST

timeZone 则是时区的设定，可以为 EST、CST、MST 和 PST 等。如果需要更多的日期和时间显示样式，则可以使用 pattern。参考 <http://java.sun.com/j2se/1.4.2/docs/api/java/text/SimpleDateFormat.html> 的说明，使用英文字母来代表各种单位。例如：yyyy 代表年份，yy 则是公元年份的后两位。使用这些字母来做组合，可以产生各种样式的时间。表 7-11 为一些简单的范例：

表 7-11

pattern	显示结果
"yyyy.MM.dd G 'at' HH:mm:ss z"	2003.02.17 公元 at 20:38:43 CST
"h:mm a"	8:35 下午
"hh 'o'clock' a, zzzz"	08 o'clock 下午, 中国标准时间
"yyMMddHHmmssZ"	030217203915+0800

最后, `var`和`scope`用来储存格式化后日期时间的属性名称和属性范围。

● `<fmt:parseDate>`

`<fmt:formatDate>`可以将字符串类型的时间或日期都转为日期时间类型。

语法

语法1: 没有本体内容

```
<fmt:formatDate value="date" [type="{time|date|both}"]
[pattern="customPattern"]
    [dateStyle="{default|short|medium|long|full}"]
    [timeStyle="{default|short|medium|long|full}"]
    [timeZone="timeZone"]
    [var="varName"]
    [scope="{page|request|session|application}"] />
```

语法2: 本体内容为欲转换为日期时间类型的字符串

```
<fmt:formatDate [pattern="customPattern"]
    [dateStyle="{default|short|medium|long|full}"]
    [timeStyle="{default|short|medium|long|full}"]
    [timeZone="timeZone"]
    [var="varName"]
    [scope="{page|request|session|application}"] />
```

属性

名 称	说 明	EL	类型	必须	默认值
value	欲被格式化的日期时间	Y	String	否	无
type	显示的部分(日期、时间或两者)	Y	String	否	date
dateStyle	设定日期部分的显示方式	Y	String	否	default
timeStyle	设定时间部分的显示方式	Y	String	否	default
pattern	格式化日期时间的样式	Y	String	否	无
timeZone	设定使用的时区	Y	String / java.util.TimeZone	否	无
parseLocale	用来取代默认的地区设定	Y	String / java.util.Locale	否	无
var	储存已格式化之日期时间	N	String	否	无
scope	var 变量的 JSP 范围	N	String	否	Page

说明

value 为欲被转换的字符串, 下面为一个简单的范例:

```
<fmt:parseDate value="2003/2/17" />
```

JSP2.0 技术手册

那么会在网页中显示 Mon Feb 17 00:00:00 CST 2003。

type 则是用来设定解析的字符串类型, 例如: 当 value 为 "08:24:06 PM", 那么我们就必须将 type 设定为 time, 如下:

```
<fmt:parseDate type="time" value="08:24:06 PM" />
```

那么就会在画面中显示 Thu Jan 01 20:24:06 CST 1970。

dateStyle 和 timeStyle 则是用来对应转换前字符串的格式。如果字符串为 "2003 年 2 月 17 日 星期一", 则必须把 dateStyle 设定成 full, 结果将会显示 Mon Feb 17 00:00:00 CST 2003。

parseLocale 和 timeZone 则是用来对应转换前字符串的地区和时区。因为某些符号在某些地区并不适用, 所以会造成无法转换的问题。例如: 台湾的表示方式是 "2003 年 2 月 17 日 星期一", 如果地区设定为 "cn", 将会出现错误无法转换。

pattern 则是对应转换前字符串的样式。例如: 当字符串为:

```
20030224T170500
```

那么我们可以使用如下方式, 将其转为日期表示方式:

```
<fmt:parseDate value="20030224T170500" pattern="yyyyMMdd'T'HHmmss" />
```

则结果将会在网页中显示 Mon Feb 24 17:05:00 CST 2003。也就是说, 可以使用 pattern 把各种特殊的字符串转换成日期格式。

最后 var 和 scope 则是用来储存转换成日期的属性名称和属性范围的。

● <fmt:setTimeZone>

<fmt:setTimeZone> 用来设定默认时区或是将时区储存至属性范围中, 方便以后使用。

语法

```
<fmt:setTimeZone value="timeZone" [var="varName"]
                    [scope="{page|request|session|application}"] />
```

属性

名 称	说 明	EL	类 型	必须	默认值
value	使用的时区	Y	String / java.util.TimeZone	是	无
var	储存设定的时区	N	String	否	无
scope	var 变量的 JSP 范围	N	String	否	Page

Null 和 错误处理

- 假若 value 为 null 或空的时, 则默认为 GMT 时区

说明

value 为设定时区，可以为 EST、CST、MST 和 PST 等。如果有 var 属性时，则会把结果存放在所设定的属性名称和属性范围中。如果没有设定 var 时，则会把它当做默认的时区，在设定的属性范围中都会套用此设定。下面的设定将会把默认时区设定为 PST，直到 session 失效为止。

```
<fmt:setTimeZone value="PST" scope="session" />
```

● <fmt:timeZone>

<fmt:timeZone>用来设定暂时的时区。

语法

```
<fmt:timeZone value="timeZone"
```

本体内容

```
</fmt:timeZone>
```

属性

名 称	说 明	EL	类 型	必须	默认值
value	使用的时区	Y	String / java.util.TimeZone	是	无

Null 和 错误处理

- 假若 value 为 null 或空的时，则默认为 GMT 时区。

说明

下面为一个简单的范例：

```
<fmt:timeZone value="PST" >
  <fmt:formatDate.../>
  <fmt:formatDate.../>
  :
</fmt:timeZone>
```

使用<fmt:timeZone>会影响到的部分只有在它的本体内容中，其他在本体内容之外将不会受到影响。

7-4 SQL 标签库 (SQL tag library)

JSTL 也提供一些与数据库有关的标签，让我们可以更轻易地查询或修改数据库的内容。但是使用这些标签时，等于是在 MVC 架构的显示层中直接存取数据库，而且它并没有 ConnectionPool 的功能，所以笔者并不建议在开发大型项目时使用这些标签。但是在一些比较

JSP2.0 技术手册

小型的网站，它还是方便好用的。

SQL 标签库主要分类如表 7-12 所示：

表 7-12

分 类	功能分类	标签名称
Database	设定	setDataSource
	SQL 指令	query dateParam paramtransaction update dateParam param

JSP 要使用 SQL 标签库时，必须先使用`<%@ taglib %>`指令，设定 SQL 标签库：

```
<%@ taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql" %>
```

7-4-1 设定

● `<sql:setDataSource>`

`<sql:setDataSource>`用来设定数据来源(DataSource)。

语法

语法1：直接使用已存在的数据来源

```
<sql:setDataSource dataSource="dataSource"
    [var="varName"]
    [scope="{page|request|session|application}"] />
```

语法2：使用JDBC方式，建立数据库联机

```
<sql:setDataSource url="jdbcUrl"
    driver="driverClassName"
    user="userName"
    password="password"
    [var="varName"]
    [scope="{page|request|session|application}"] />
```

属性

名 称	说 明	EL	类型	必须	默认值
dataSource	数据来源	Y	String / javax.sql.DataSource	否	无
driver	JDBC 驱动程序的类名称	Y	String	否	无
url	数据库的 URL	Y	String	否	无

JSP2.0 技术手册

续表

名 称	说 明	EL	类型	必须	默认值
user	用户名称	Y	String	否	无
password	用户密码	Y	String	否	无
var	储存数据来源	N	String	否	无
scope	var 变量的 JSP 范围	N	String	否	Page

说明

当我们要使用 JDBC 连接到数据库时，通常需要有数据库的 JDBC 驱动程序、URL、用户名称和密码。假设我们现在有一个 MySQL 的数据库，它所需信息如下：

- (1) 用户名称：root
- (2) 用户密码：root
- (3) jdbc 驱动程序类名称：com.mysql.jdbc.Driver
- (4) 数据库 URL：jdbc:mysql://localhost/test

那么当我们使用<sql:setDataSource>来设定数据来源时，写法必须如下：

```
<sql:setDataSource driver="com.mysql.jdbc.Driver"
    user="root"
    password="root"
    url="jdbc:mysql://localhost/test" />
```

因为我们没有设定 var 和 scope 这两个属性，因此它会取代掉原有在 Page 属性范围的默认数据来源。如果设定了 scope 和 var 这两个属性，则会把数据来源储存至 varName 之中。

如果是 JNDI，假设我们在 Tomcat 的 server.xml 和 web.xml 中设定好 JNDI 的名称为 jdbc/TestDB，则可以写成如下：

```
<sql:setDataSource dataSource="java:comp/env/jdbc/TestDB" />
```

那么就会自动寻找所有属性范围中名称为 myDB 的 DataSource 对象。

如果是 JNDI，假设我们在 server.xml 和 web.xml 中已设定好 JNDI 名称为 jdbc/TestDB，则可以写成如下：

```
<sql:setDataSource dataSource=" jdbc/TestDB" />
```

如果为JDBC参数字符串，则必须符合格式url[, [driver][, [user][, password]]]，所以可以写成如下：

```
<sql:setDataSource
dataSource="jdbc:mysql://localhost/test,com.mysql.jdbc.Driver,root,root"/>
```

将数据来源设定好以后，就可以查询或修改数据库的内容了。下面开始笔者将依序介绍其他有关数据库功能的标签。

7-4-2 SQL 指令

此分类中包含五个标签，它们分别为：<sql:query>、<sql:update>、<sql:param>、<sql:dateparam>和<sql:transaction>，接下来依序介绍如何使用这五个标签。

● <sql:query>

<sql:query>用来查询数据库的数据。

语法

语法1：没有本体内容

```
<sql:query sql="sqlQuery" var="varName"
    [scope="{page|request|session|application}"]
    [dataSource="dataSource"]
    [maxRows="maxRows"]
    [startRow="startRow"] />
```

语法2：本体内容为查询指令

```
<sql:query var="varName"
    [scope="{page|request|session|application}"]
    [dataSource="dataSource"]
    [maxRows="maxRows"]
    [startRow="startRow"]>
....
sqlQuery .....
</sql:query>
```

属性

名 称	说 明	EL	类 型	必 须	默认值
sql	SQL 语法(Select ...)	Y	String	否	无
dataSource	数据来源	Y	String / javax.sql.DataSource	否	无
maxRows	设定最多可暂存的数据笔数	Y	String	否	无
startRow	设定数据从第几笔开始	Y	String	否	0
var	储存查询结果	N	String	是	无
scope	var 变量的 JSP 范围	N	String	否	Page

注意

var 属性在<sql:query>是必须的，因为查询后的结果都必须储存至 varName 中。

说明

首先我们使用<sql:query>时必须指定操作的对象，所以 dataSource 要指定好数据来源。

假设我们使用<sql:setDataSource>把数据来源储存在属性范围为 Session 的 myDB 中, 那么使用<sql:query>操作此数据来源时写法如下:

```
<sql:query dataSource="${sessionScope.myDB}" ...../>
```

另外, dataSource 属性是可以省略的, 当我们省略此属性时, <sql:query>操作的对象将会是同一 Page 属性范围中的默认数据来源, 假若找不到, 则会产生 JspException 的错误。

我们可以使用 SQL 语法来查询数据库的内容, 写法有两种: 一种是把 SQL 语法写在<sql:query>的本体内容中; 另一种是把 SQL 语法写在<sql:query>的 sql 属性之中。写法如下:

```
<sql:query sql="SELECT * FROM test" var="result" />
```

或者是

```
<sql:query var="result">
SELECT * FROM test
</sql:query>
```

此时, 网页中不会显示任何东西, 只会将查询结果存放在属性范围中。以上述的范例来说, 将会把结果存放在 result 的属性中, 因为我们没有指定属性范围, 所以默认为 Page。varName 存放所有查询结果和信息的对象, 它总共有 5 个属性, 如表 7-13 所示:

表 7-13

属性名称	内 容
rows	以字段名称当做索引的查询结果
rowsByIndex	以数字当做索引的查询结果
columnNames	字段名称
rowCount	查询到的数据笔数
limitedByMaxRows	取出最大数据笔数的限制

在这里笔者用实际的例子来介绍这几个属性。假设笔者现在有一数据库, table 名称为 test, 内容如表 7-14 所示:

表 7-14

Id	Age
Barry	25
Joe	51
Hellen	16
Hwang	32

当<sql:query>写法如下时, 将会把这四笔数据和一些相关信息存放到 result 中。

```
<sql:query var="result">
SELECT * FROM test
</sql:query>
```

接下来可以使用\${result.columnNames[0]}和\${result.columnNames[1]}来取出全部的字

段名称, 使用`${result.rowCount}`得到查询的数据笔数, 因此查询结果的写法为:

```
<table>
<c:forEach items="${result.rows}" var="row" >
<tr>
<td>${row.Id}</td>
<td>${row.Age}</td>
</tr>
</c:forEach>
</table>
```

或者是

```
<table>
<c:forEach items="${result.rowsByIndex}" var="row" >
<tr>
<td>${row[0]}</td>
<td>${row[1]}</td>
</tr>
</c:forEach>
</table>
```

以上两种写法所产生的结果都是相同的, 也就是说, 可以利用字段名称或者数字当做索引。因为 `rows` 和 `rowsByIndex` 这两个属性都存放在查询结果之中, 所以我们可以使用 `<c:forEach>` 来依序浏览它的成员。`limitedByMaxRows` 则要和 `<sql:query>` 的 `maxRows` 一起使用。

`<sql:query>` 的 `startRow` 主要用来设定将符合查询的数据从第几笔开始放到查询结果中, 而 `maxRows` 则是设定最多可以放入几笔数据。`startRow` 是以 0 为第一笔数据, 以此类推。范例如下:

```
<sql:query var="result" maxRows="2" startRow="1">
SELECT * FROM test
</sql:query>
<table>
<c:forEach items="${result.rows}" var="row" >
<tr>
<td>${row.Id}</td>
<td>${row.Age}</td>
</tr>
</c:forEach>
</table>
```

我们把查询结果设定成从第二笔数据开始, 总共两笔数据。那么网页中显示的查询结果会是:

```
Joe 51
Hellen 16
```

查询结果的 `limitedByMaxRows` 则用来判断是否因为 `maxRows` 的大小而受到限制。如果查询到的数据笔数大于 `maxRows` 时, 那么 `limitedByMaxRows` 将会是 `true`; 如果为等于或者是小于 `maxRows`, 则表示没有受到 `maxRows` 的大小限制, 那么结果会是 `false`。

● `<sql:update>`

`<sql:update>` 用来修改数据库的数据。

语法

语法1: 没有本体内容

```
<sql:query sql="sqlUpdate" [var="varName" ]
           [scope="{page|request|session|application}"]
           [dataSource="dataSource"] />
```

语法2: 本体内容为查询指令

```
<sql:query [var="varName"] [scope="{page|request|session|application}"]
           [dataSource="dataSource"] >
```

```
....
sqlUpdate .....
</sql:query>
```

属性

名 称	说 明	EL	类 型	必 须	默认值
sql	SQL 语 法 (Update 、 Insert 、 Delete ...)	Y	String	否	无
dataSource	数据来源	Y	String / javax.sql.DataSource	否	无
var	储存改变的数据笔数	N	String	否	无
scope	var 变量的 JSP 范围	N	String	否	Page

说明

这几个属性基本上和<sql:query>相似, 在<sql:update>中我们可以使用 UPDATE、INSERT 和 DELETE 这几个 SQL 语法。我们可以把 SQL 语法写在<sql:query>的本体内容中; 或是写在<sql:query>的 sql 属性中, 如下:

```
<sql:update>
INSERT INTO test(Id, Age)
VALUES('John', 34)
</sql:update>
```

也可以写在<sql:update>的 sql 属性中, 如下:

```
<sql:update sql="INSERT INTO test(Id, Age) VALUES('John', 34)"/>
```

var 则是设定属性名称, 用来储存改变的数据笔数; scope 则是用来设定 var 的范围。假设我们要从数据库中删除数据, 代码如下:

```
<sql:update var="n">
DELETE FROM test
WHERE Age > 30
</sql:update>
```

JSP2.0 技术手册

n 中存放的就是被删除掉的数据笔数，如果为 0，则代表没有删除任何一笔数据。如果是 UPDATE 指令时，n 则是代表被修改的数据笔数；如果 SQL 指令是 INSERT 时，则 n 就代表新增的数据笔数。

● <sql:param>和<sql:dateParam>

<sql:param>和<sql:dateParam>让 SQL 语句能动态设定变量。

语法

• <sql:param>

语法1:

```
<sql:param value="value" />
```

语法2:

```
<sql:param>
value
</sql:param>
```

• <sql:dateParam>

```
<sql:dateParam value="value" [type="{timestamp|time|date}"] />
```

属性

• <sql:param>

名 称	说 明	EL	类 型	必须	默认值
value	Object 类型的参数	Y	Object	否	无

• <sql:dateParam>

名 称	说 明	EL	类 型	必须	默认值
value	Date 类型的参数	Y	java.util.Date	是	无
type	Date 的种类(timestamp、time 或 date)	Y	String	否	timestamp

说明

假若我们的 SQL 指令需要一些动态变量时，可以写成如下方式：

```
<sql:query var="result">
SELECT * FROM test
WHERE Id='${userId}' />
</sql:query>
```

但是上述写法将会发生一些问题，例如：当 userId 的内容为 root' OR Id <> 'root 时，SQL 的查询语句为：

```
SELECT * FROM test WHERE Id='root' OR Id <> 'root'
```

执行结果会将整个数据库的内容都显示出来，因为任一笔数据一定都会符合上述条件。因此，`<sql:param>`和`<sql:dateParam>`提供动态给予参数的方式来防止类似上述的问题。

首先看一下`<sql:param>`的 `value` 就是放入数据库的动态参数，每一个`<sql:param>`都必须存在于`<sql:update>`或`<sql:query>`之中，且对应到 SQL 语法中的每一个“?”。假设我们要限定某个条件来查询数据，可以写成如下形式：

```
<sql:query var="result">
SELECT * FROM test
WHERE Id=?
AND Age > ?
<sql:param value="${param.Id}" />
<sql:param value="${param.Age}" />
</sql:query>
```

这样会以用户输入的参数当做条件，来查询数据库中是否有符合的数据。

`<sql:param>`可用于各种数据格式，但是当参数为有关时间或日期时，则必须使用`<sql:dateParam>`来设定 SQL 语法中的 `java.util.Date` 类型。

`<sql:dateParam>`用法和`<sql:param>`相同，不一样的地方只在于它多一 `type` 属性。`type` 是用来指定数据库处理参数的类型。如果存放的是时间就使用 `time`；日期则用 `date`；而时间和日期一起，则用 `timestamp`。

● `<sql:transaction>`

`<sql:transaction>`提供存取数据库时的一种安全的机制。

语法

```
<sql:transaction [dataSource="dataSource"]
    [isolation="{read_committed|read_uncommitted|repeatable|serializable}"] >
<sql:query> or <sql:update>
</sql:transaction>
```

属性

名 称	说 明	EL	类 型	必须	默认值
dataSource	数据来源	Y	String / javax.sql.DataSource	否	无
isolation	交易互不干扰等级	Y	String	否	无

说明

当同一时间查询、修改数据的动作大量增加时，很容易就会发生数据不一致的问题。例如：

JSP2.0 技术手册

从某个账户转账到另一个账户，如果扣完钱以后服务器突然当掉，那么这笔钱从第一个账户扣款，但却没有转到另一个账户，此时将会使得这笔钱凭空消失。因此，应提供一些安全机制来保护这些交易的行为。

`<sql:transaction>`主要是将所有必须同时执行的交易放在它的本体内容中。如下范例：

```
<sql:transaction>
  <sql:update...>
  :
  <sql:query....>
  :
  <sql:update...>
  :
</sql:transaction>
```

如果当`<sql:transaction>`的本体内容有错误发生时，将不会执行任何一个 SQL 语句，所以可以保障交易机制的安全性。

`isolation` 属性主要是让我们设定交易的安全等级，让我们可以预防许多常见的错误。JDBC 支持了四种交易模式，而`<sql:transaction>`也提供此四种功能，如表 7-15：

表 7-15

互不干扰等级	Dirty reads?	Nonrepeatable reads?	Phantom reads?
read_uncommitted	-	-	-
read_committed	预防	-	-
repeatable_read	预防	预防	-
serializable	预防	预防	预防

当我们在数据上设定一个范围的锁定(lock)后，其他的用户在对此数据做查询或修改的动作时，能够确保数据的一致性。以下是实现方式：

```
<sql:transaction isolation="serializable">
  <sql:update...>
  :
  <sql:query....>
  :
  <sql:update...>
  :
</sql:transaction>
```

7-5 XML 标签库 (XML tag library)

JSTL 提供了一些有关 XML 的标签，让我们可以不用深入了解 SAX 或 DOM 等 API，就可以轻易地处理 XML 文件。这几个标签依照功能主要可分为两大类：(1) 对 XML 文件做剖析、

显示其中部分内容，或者是加入一些判断式来判断某些内容是否存在；(2) 让 XML 文件搭配 XSLT 显示在 JSP 页面中。笔者将 JSTL 中有关 XML 的处理分为三类如表 7-16 所示：

表 7-16

分 类	功能分类	标签名称
XML	核心操作	out parse set
	流程控制	choose when otherwise forEach if
	文件转换	transform param

在介绍这些标签之前，首先必须学习一些有关 XPath 的概念，因为这些标签都使用到了 XPath 的语句，因此笔者只是简单介绍 XPath 的概念，读者如对 XPath 有兴趣，可以自行去翻阅 XML 的教学书籍。

7-5-1 XPath

XPath，简单来说就像是 UNIX 下的目录。我们搭配范例一起说明，底下是一个简单的 XML 文件 *Player.xml*：

```
■ Player.xml
<?xml version="1.0" encoding="GB2312"?>
<选手 no="23">
  <姓名>
    <姓>迈克尔</姓>
    <名>乔丹</名>
  </姓名>
  <位置>前锋</位置>
  <球队>巨人</球队>
</选手>
```

首先来介绍一些专有名词：

■ 根元素

在任何一个 XML 文件都会有的起始位置，也就是 XML 结构树的 root 部分。在 XPath 中必须用 “/” 表示。

■ 文件元素

位于 root 底下的第一个元素，每一个 XML 文件只能有一个文件元素。在这个范例中：“选手”就是文件元素。

■ 父子节点，兄弟节点

父节点也就是比现在节点高一级别的节点，子节点就是比现在节点低一级别的节点。而兄弟节点表示在同一级别中的节点。在上面 *Player.xml* 中：

选手 是 姓名 的父节点

姓 是 姓名 的子节点

姓名 是 球队 的兄节点

位置 是 球队 的弟节点

接下来我们要用 XPath 来表示 XML 的内容，可以标记成如表 7-17：

表 7-17

位 置	表示方式
根元素	/
文件元素	./
父节点	../
子节点	节点名称
兄弟节点	../节点名称
所有同一节点名称	//节点名称
属性	@属性名称
全局	*

如果要用 XPath 表示上面的 *Player.xml*，则可以写成如表 7-18 的形式：

另外比较特别的是，在 JSTL 中使用 XPath 也可以搭配 EL 的隐含对象和表达式，在后面的范例中将会多加说明。

表 7-18

位 置	表示方式
文件元素	/选手
姓名节点	/选手/姓名
位置节点	/选手/位置
球队节点	/选手/球队
姓节点	/选手/姓名/姓
名节点	/选手/姓名/名
no 的属性值	/选手/@no

续表

位 置	表示方式
所有名节点	//名
姓名节点的所有子节点	/选手/姓名/*

7-5-2 核心操作

- **<x:parse>**
<x:parse>用来解析 XML 文件。

语法

```
语法1:
<x:parse doc="XMLDocument"
    {var="var" [scope="{page|request|session|application}"]}
    {varDom="var" [scope="{page|request|session|application}"]} }
    [systemId="systemId"]
    [filter="filter"] />

语法2:
<x:parse {var="var" [scope="{page|request|session|application}"]}
    {varDom="var" [scope="{page|request|session|application}"]} }
    [systemId="systemId"]
    [filter="filter"] >

欲被解析的XML文件
</x:parse>
```

属性

名 称	说 明	EL	类 型	必须	默认值
doc	XML 文件	Y	String / Reader	否	无
systemId	XML 文件的 URI	Y	String	否	无
filter	XMLFilter 过滤器	Y	org.xml.sax.XMLFilter	否	无
varDom	储存解析后的 XML 文件(类型为 org.w3c.dom.Document)	N	String	否	无
scopeDom	varDom 的范围	N	String	否	Page
var	储存解析后的 XML 文件	N	String	否	无
scope	var 变量的 JSP 范围	N	String	否	Page

说明

doc 属性主要设定要被解析的 XML 文件地址,或者是将它写在<x:parse>的本体内容之中。通常我们会搭配<c:import>使用,笔者在撰写时,使用 <http://www.jsptw.com/jute/MyXML.xml> 来当做

范例。底下为搭配<c:import>的使用范例。

```
<c:import var="sample" url="http://www.jsptw.com/jute/MyXML.xml" />
<x:parse doc="{sample}" var="sampleXml" />
```

我们也可以把它放在<x:parse>的本体内容中。如下：

```
<x:parse doc="{sample}" var="sampleXml" />
<c:import url="http://www.jsptw.com/jute/MyXML.xml" />
</x:parse>
```

比较特别的是 var 这个属性必须存在，用来设定属性名称，以后有关 XML 的标签都会用到 var 所设定的名称；scope 则用来设定属性范围。

● <x:out>

<x:out>主要用来取出 XML 中的字符串。

语法

```
<x:out select="XPathExpression" [escapeXml="{true|false}"] />
```

属性

名 称	说 明	EL	类 型	必须	默认值
select	XPath 语句	N	String	否	无
escapeXml	是否转换特殊字符，例如：< 转换成 <	Y	boolean	是	true

说明

select 主要放在我们要执行的 XPath 中，这时就会用到在<x:parse>里 var 所设定的名称。这里介绍一个简单的范例，教导读者如何使用<x:out>和它的 XPath 语句。

```
<x:parse var="sampleXml">
  <item>
    <name>car</name>
    <price>10000</price>
  </item>
</x:parse>
```

假若我们要显示<name>的内容时，写法可以有以下两种：

```
<x:out select="$sampleXml//name" />
```

或者是

```
<x:out select="$sampleXml/item/name" />
```

escapeXml 则设定是否将内容中的特殊字符转换，例如：将 < 转换成 <。默认为 true。

● <x:set>

<x:set>将从 XML 文件取得的内容储存至属性范围中。

语法

```
<x:set select="XPathExpression" var="var"
      [scope="{page|request|session|application}"] />
```

属性

名 称	说 明	EL	类 型	必须	默认值
select	XPath 语句	N	String	是	无
var	将从 XML 文件中取得的内容储存至 varName 中	N	String	是	无
scope	var 变量的 JSP 范围	N	String	否	Page

说明

这里笔者直接使用范例来介绍它的用法：

```
<x:parse var="sampleXml">
  <items>
    <item>
      <name>car</name>
      <price>10000</price>
    </item>
  </items>
</x:parse>
```

那么当我们要把某部分内容抓出来时可以写成：

```
<x:set var="price" select="$sampleXml/items/item/price" />
```

它会直接把 10000 取出来，存入 price 变量当中。我们可以使用<c:out value="{price}" />将它显示在页面上。如果要抓的是 XML 文件，则可以写成：

```
<x:set var="item" select="$sampleXml//item" />
```

那么它就会取出如下的数据：

```
<item>
  <name>car</name>
  <price>10000</price>
</item>
```

即取出 XML 中的部分内容，读者可以自行使用<c:out>显示，结果将会是[[item:null]]而不是上述的数据内容。如果要显示 10000 时，则可以使用：

```
<x:out select="$item/price" />
```

7-5-3 流程控制

在介绍<x:parse>、<x:out>和<x:set>之后，下面将要介绍 XML 中流程控制的部分。流程控制功能主要分为两类：

- (1) 条件判断，在 JSP 中当我们想要依据 XML 的内容做某些动作时，条件判断部分就会使我们处理起来方便许多。
- (2) 循环功能，让整个 XML 文件被循序浏览一遍。

当然<x:if>和<x:choose>就是所谓的条件判断，而<x:forEach>就是循环功能。

● <x:if>

<x:if>和<c:if>类似，只是条件判断的内容为 XPath 语句。

语法

语法1：无本体内容

```
<x:if select="XPathExpression" var="varName"
    [scope="{page|request|session|application}"] />
```

语法2：有本体内容

```
<x:if select="XPathExpression" [ var="varName"
    [scope="{page|request|session|application}"] />
```

本体内容

```
</x:if>
```

属性

名 称	说 明	EL	类 型	必 须	默认值
select	XPath 语句，如果为 true，则执行<x:if>的本体内容	N	String	是	无
var	设定属性名称存放判断结果(true 或 false)	N	String	否	无
scope	var 变量的 JSP 范围	N	String	否	Page

说明

假设现在有一份顾客数据的 XML 文件，它包含多个 customer 元素，每个 customer 元素都有一个 id 属性，而每个 customer 元素里面又有一个 name 元素和多个或零个 phone 元素。phone 元素表示顾客留过电话数据。底下为一个简单的范例：

```
<customers>
  <customer id="1">
    <name>brower</name>
    <phone>156-1324</phone>
    <phone>201-9999</phone>
  </customer>
  <customer id="2">
```

JSP2.0 技术手册

```

    <name>koji</name>
  </customer>
</customers>

```

现在我们有一份网页要询问用户是否曾经留过电话。如果留过，将显示一段文字，表示当有特殊活动时会用电话通知用户。所以，我们要写一个判断式用来判断用户是否留过电话。判断方式可以写成如下：

```

<x:if select="$doc/customers/customer
[@id=$requestScope:customerId]/phone">
  感谢您留下您的个人电话，当有特殊活动时我们将会电话通知您
</x:if>

```

`$doc/customers/customer [@id=$requestScope:customerId]/phone` 的意思为：当 `customer` 元素的 `id` 等于 `requestScope` 的 `customerId`，且其 `customer` 元素中包含 `phone` 元素时才会显示 `<x:if>` 的本体内容。

● `<x:choose>`、`<x:when>`和`<x:otherwise>`

XML 标签库提供 `<x:choose>`、`<x:when>`和`<x:otherwise>`，功能和核心标签库的类似，只是 `<x:when>`使用的是 XPath 语句。除了 `<x:when>`有一个属性需要设定外，其他两个标签都没有属性。

属性

名 称	说 明	EL	类 型	必 须	默认值
select	XPath 语句，如果为 true，则执行 <code><x:when></code> 的本体内容	N	String	是	无

说明

我们一样用 `<x:if>` 中的 XML 数据来做范例，在 `customer` 元素中多了一个属性 `status`，代表此顾客的等级。底下为一个简单的范例：

```

<customers>
  <customer id="1" status="vip">
    <name>browser lin</name>
  </customer>
  <customer id="2" status="normal">
    <name>koji lin</name>
  </customer>
  <customer id="3" status="new">
    <name>mei huang</name>
  </customer>
</customers>

```

现在要判断顾客的等级并给予不同的信息。判断方式可以写成如下：

```

<x:choose>
  <x:when select="$doc//customer[@id=$customerId]/@status='vip'">
    亲爱的主任您好
  </x:when>

```

```

<x:when select="$doc//customer[@id=$customerId]/@status='normal'">
  亲爱的客户您好
</x:when>

<x:otherwise>
  您好
</x:otherwise>
</x:choose>

```

`$doc//customer[@id=$customerId]/@status='normal'` 的意思为：依据不同的 status，我们可以给予各种相对应的结果。例如：当指定的 customerId 元素的 customer 中，属性 status 的值为 vip 时就显示“亲爱的主任您好”。

● <x:forEach>

<x:forEach> 和 <c:forEach> 类似，不过 <c:forEach> 浏览的是各种 java 集合对象，而 <x:forEach> 浏览的是 xml 文件的内容。

语法

```

<x:forEach select="XPathExpression" [var="varName"] 劳任怨
  varStatus="varStatusName" [begin="begin"] [end="end"] [step="step"]>
  本内容
</x:forEach>

```

属性

名 称	说 明	EL	类 型	必须	默认值
var	设定变量储存现在浏览到的节点	N	String	否	无
select	XPath 语句，决定被浏览的部分	N	String	是	无
varStatus	用来存放现在指到的相关成员信息	N	String	否	Page
begin	开始的位置	Y	int	否	无
end	结束的位置	Y	int	否	无
step	每次迭代的间隔数	Y	int	否	无

说明

当我们想把 XML 中某个元素名称中的数据全部显示出来时，就可以使用 <x:forEach>，例如：当我们的语句为 `//customer`，那么它将会浏览 XML 中所有的 customer 元素。

底下为一个简单的范例：

```

<x:parse var="doc">
<customers>
  <customer id="1" status="vip">
    <name>browner lin</name>
  </customer>
</customers>
</x:parse>

```

JSP2.0 技术手册

```

</customer>
<customer id="2" status="normal">
  <name>koji lin</name>
</customer>
<customer id="3" status="new">
  <name>mei huang</name>
</customer>
</customers>
</x:parse>

<x:forEach select="$doc//customer">
<p>找到一笔数据</p>
</x:forEach>

```

那么将会在网页中显示三次<p>找到一笔数据</p>。如果我们想在网页中显示每个数据的内容,则写法如下:

```

<x:forEach select="$doc//customer">
<p><x:out select="name" /></p>
</x:forEach>

```

之前曾经提到过,假若我们要显示内容时,必须使用完整的 XPath 语句 \$doc/customer/name。但是,这里要显示的 name 元素都在现在指到的 customer 元素底下,因此我们使用类似相对位置的概念,直接利用 name 来取得内容。

7-5-4 文件转换

● <x:transform>

<x:transform>让我们能轻易地使用 XSLT 重新包装 xml 文件,成为另外一种显示的方式。

语法

语法1: 没有本体内容

```

<x:transform doc="XMLDocument" xslt="XSLTStylesheet"
  [docSystemId="XMLSystemId"]
  [xsltSystemId="XSLTSystemId"]
  [{var="varName" [scope="scopeName"]|result="resultObject"}] />

```

语法2: 本体内容有<x:param>

```

<x:transform doc="XMLDocument" xslt="XSLTStylesheet"
  [docSystemId="XMLSystemId"]
  [xsltSystemId="XSLTSystemId"]
  [{var="varName" [scope="scopeName"]|result="resultObject"}] >
<x:param> ...
</x:transform>

```

语法3: 本体内容有<x:param>和XML文件

```

<x:transform xslt="XSLTStylesheet" [docSystemId="XMLSystemId"]
  [xsltSystemId="XSLTSystemId"]
  [{var="varName" [scope="scopeName"]|result="resultObject"}] >

```

JSP2.0 技术手册


```
XML文件 ...
<x:param> ...
</x:parse>
```

属性

名 称	说 明	EL	类 型	必须	默认值
doc	XML 文件	Y	String	否	无
xslt	要套用的 XSLT 样式	Y	String	是	无
docSystemId	XML 文件的 URI	Y	String	否	无
xsltSystemId	XSLT 文件的 URI	Y	String	否	无
var	存放转换后的文件	N	String	否	无
scope	var 变量的 JSP 范围	N	String	否	Page
result	转换结果的对象	Y	javax.xml.transform.Result	否	无

说明

doc 属性是被转换的 XML 文件；xslt 是要套用的 XSLT 样式，这两种文件都必须先指定好。假若 doc 属性不存在时，必须从<x:transform>的本体内容中取得 XML 文件，我们有时也会搭配<c:import>写成：

```
<c:import var="xmlDocument" url="${xmlUrl}" />
<c:import var="xsltStylesheet" url="${xsltUrl}" />
<x:transform doc="${xmlDocument}" xslt="${xsltStylesheet}" />
```

url 中的 xmlUrl 和 xsltUrl 就是前面提到的两个文件。底下为一个简单的范例：

■ X_transform.jsp

```
<%@ page import = "java.io.*,java.sql.*,javax.sql.*" %>
<%@ page contentType="text/html;charset=GB2312" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="x" uri="http://java.sun.com/jsp/jstl/xml" %>

<c:set var="xml">
  <document>
    <header>XML 搭配 XSLT</header>
    <paragraph>让我们把 xml 用 xslt 转换后显示在网页上</paragraph>
  </document>
</c:set>

<c:set var="xsl">
  <?xml version="1.0" encoding="GB2312" ?>
  <xsl:stylesheet version=
    "1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform" >
    <xsl:template match="/">
      <xsl:apply-templates/>
    </xsl:template>
    <xsl:template match="document">
```

```

<html>
  <title>CH7 - X_transform.jsp</title>
  <h2>&lt;x:transform&gt; 的使用</h2>
  <xsl:apply-templates/>
</html>
</xsl:template>
<xsl:template match="header">
  <h1><xsl:value-of select="."/></h1>
</xsl:template>
<xsl:template match="paragraph">
  <p><xsl:value-of select="."/></p>
</xsl:template>
</xsl:stylesheet>
</c:set>
<x:transform doc="${xml}" xslt="${xsl}" />

```

我们可以先将<c:set>的部分存成文件，然后再用<c:import>来取得。另外 var 和 scope 则用来设定属性名称和范围，储存转换后的文件。

另外，如果想把 XML 文件一层一层套用 XSTL 样式，那么可以写成：

```

<x:transform var="temp1" xml="${xml}" xslt="${xsl1}"/>
<x:transform var="temp2" xml="${temp1}" xslt="${xsl2}"/>
<x:transform xml="${temp2}" xslt="${xsl3}"/>

```

或者是

```

<x:transform xslt="${xsl3}">
  <x:transform xslt="${xsl2}">
    <x:transform xslt="${xsl1}" xml="${xml}" />
  </x:transform>
</x:transform>

```

X_transform.jsp 的执行结果如图 7-21。



图 7-21 X_transform.jsp 的执行结果

● <x:param>

<x:param>用来设定转换(transform)参数。

语法

语法1: value属性设定参数值

```
<x:param name="name" value="value" />
```

语法2: 本体内容为参数值

```
<x:param name="name">
parameter value
</x:param>
```

属性

名 称	说 明	EL	类 型	必须	默认值
name	参数名称	Y	String	是	无
value	参数值	Y	Object	无	无

说明

`<x:param>`只能在`<x:transform>`中使用,例如:当我们有一参数名称为 music,值为 classic 时,可以写成:

```
<x:transform>
.....
<x:param name="music" value="classis" />
.....
</x:transform>
```

或者是

```
<x:transform>
.....
<x:param name="music">
classic
<x:param>
.....
</x:transform>
```

7-6 函数标签库 (Functions tag library)

函数标签库是 JSTL 1.1 版新增的,不过严格说起来,它不能算是标签库,因为它是利用 EL 的 Function 所实现出来的。这部分笔者已在第六章的“EL Functions”中说明了它的实现方式。

函数标签库大部分都是用来处理字符串用的,以下是字符串功能的函数(见表 7-19):

JSP2.0 技术手册

表 7-19

fn:contains()	fn:join()	fn:startsWith()	fn:toUpperCase()
fn:endsWith()	fn:split()	fn:substring()	fn:toLowerCase()
fn:escapeXml()	fn:trim()	fn:substringBefore()	fn:containsIgnoreCase()
fn:indexOf()	fn:replace()	fn:substringAfter()	

除了处理字符串的函数之外，函数标签库还有 fn:length 函数，它用来取得字符串的字符数或集合对象的大小。

在 JSP 中要使用 JSTL 中的函数标签库时，必须先在 `<%@ taglib %>` 指令中设定 prefix 和 uri 的值，一般设定如下：

```
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
```

说明

笔者认为函数标签库应在第六章的“EL Functions”中说明介绍，不过因为 JSTL 1.1 的规范书中有一个完整的章节介绍它，所以笔者还是将它放在 JSTL 章节中说明。

● <fn:contains>

判断某字符串是否在一字符串之中。

语法

```
{fn:contains(string, substring)} => boolean
```

属性

名 称	说 明	类 型
string	原输入的字符串	String
substring	测试用的字符串	String
Result	回传 string 是否包含 substring。若有，则回传 true；若无，则回传 false	boolean

Null 和 错误处理

- 假若 string 为 null 时，它会以空字符串做处理
- 假若 substring 为 null 时，它会以空字符串做处理

说明

`{fn:contains(string, substring)}` 函数用来判断 string 字符串是否包含 substring 字符串。

范例

■ Fn_contains.jsp

```
.....
```

```
<c:set var="s1" value="There is a castle on a cloud"/>
```

JSP2.0 技术手册

```

${fn:contains(s1, "castle")}
${fn:contains(s1, "CASTLE")}
${fn:contains(s1, undefined)}
${fn:contains(s1, "")}
${fn:contains(undefined, "castle")}
${fn:contains(undefined, "")}
.....

```

执行结果如图 7-22 所示。

第二个结果为 false, 可知 fn:contains() 函数是考虑大小写的。假若要忽略大小写时, 就要改用 fn:containsIgnoreCase() 函数。

● <fn:containsIgnoreCase>

判断某字符串是否在已有字符串之中, 并忽略其大小写。

Input String	Substring	Result
There is a castle on a cloud	castle	true
There is a castle on a cloud	CASTLE	false
There is a castle on a cloud	null	true
There is a castle on a cloud	empty string	true
null	castle	false
null	empty string	true

图 7-22 Fn_contains.jsp 的执行结果

语法

```
${fn:containsIgnoreCase(string, substring)} => boolean
```

属性

名 称	说 明	类型
string	原输入的字符串	String
substring	测试用的字符串	String
Result	回传 string 是否包含 substring, 并且忽略大小写。若有, 回传 true; 若无, 则回传 false	boolean

Null 和 错误处理

同<fn:contains>

说明

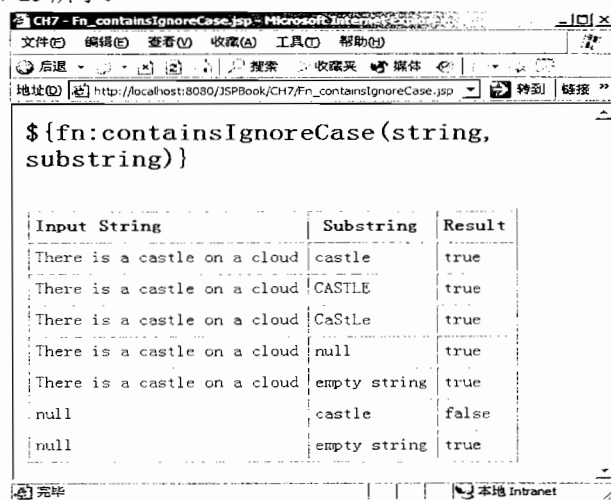
它和`{fn:containsIgnoreCase(string, substring)}`函数的功能一样,只不过它忽略大小写。

范例

■ `Fn_containsIgnoreCase.jsp`

```
.....  
<c:set var="s1" value="There is a castle on a cloud"/>  
  
{fn:containsIgnoreCase(s1, "castle")}  
{fn:containsIgnoreCase(s1, "CASTLE")}  
{fn:containsIgnoreCase(s1, "CaStLe")}  
{fn:containsIgnoreCase(s1, undefined)}  
{fn:containsIgnoreCase(s1, "")}  
{fn:containsIgnoreCase(undefined, "castle")}  
{fn:containsIgnoreCase(undefined, "")}  
.....
```

执行结果如图 7-23 所示。



Input String	Substring	Result
There is a castle on a cloud	castle	true
There is a castle on a cloud	CASTLE	true
There is a castle on a cloud	CaStLe	true
There is a castle on a cloud	null	true
There is a castle on a cloud	empty string	true
null	castle	false
null	empty string	true

图 7-23 `Fn_containsIgnoreCase.jsp` 的执行结果

● `<fn:startsWith>`

判断一字符串是否以某一字符串为开头。

语法

```
${fn:startsWith(string, prefix)} => boolean
```

属性

名 称	说 明	类型
string	原输入的字符串	String
prefix	测试用的字符串	String
Result	回传 string 是否以 prefix 字符串开头。若是，回传 true；若不是，则回传 false	boolean

Null 和 错误处理

同<fn:contains>

说明

`${fn:startsWith(string, prefix)}` 函数用来判断 string 字符串是否以 prefix 字符串开头。

范例

■ Fn_startsWith.jsp

```
.....
<c:set var="s1" value="There is a castle on a cloud"/>

${fn:startsWith(s1, "castle")}
${fn:startsWith(s1, "There is")}
${fn:startsWith(s1, undefined)}
${fn:startsWith(s1, "")}
${fn:startsWith(undefined, "castle")}
${fn:startsWith(undefined, "")}
.....
```

执行结果如图 7-24 所示。

● <fn:endsWith>

判断一字符串是否以某一字符串为结尾。

语法

```
${fn:endsWith(string, suffix)} => boolean
```

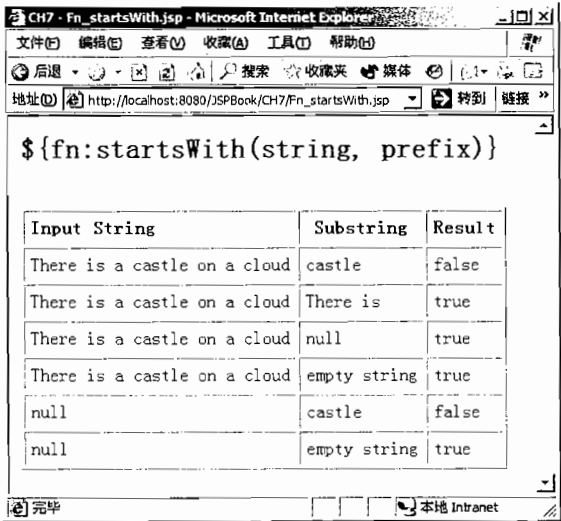


图 7-24 Fn_startsWith.jsp 的执行结果

属性

名 称	说 明	类 型
string	原输入的字符串	String
suffix	测试用的字符串	String
Result	回传 string 是否以 suffix 字符串结尾。若是，回传 true；若不是，则回传 false	boolean

Null 和 错误处理

同<fn:contains>

说明

`<fn:endsWith(string, suffix)>` 函数用来判断 string 字符串是否以 suffix 字符串为结尾。

范例

■ Fn_endsWith.jsp

```
.....
<c:set var="s1" value="There is a castle on a cloud"/>

${fn:endsWith(s1, "castle")}
${fn:endsWith(s1, "cloud")}
${fn:endsWith(s1, undefined)}
${fn:endsWith(s1, "")}
```

JSP2.0 技术手册

```

${fn:endsWith(undefiend, "castle")}
${fn:endsWith(undefiend, "")}
.....

```

执行结果如图 7-25 所示。

● <fn:escapeXml>

转义字符将被转换成 Entity 码。

语法

```

${fn:escapeXml(string)} => String

```

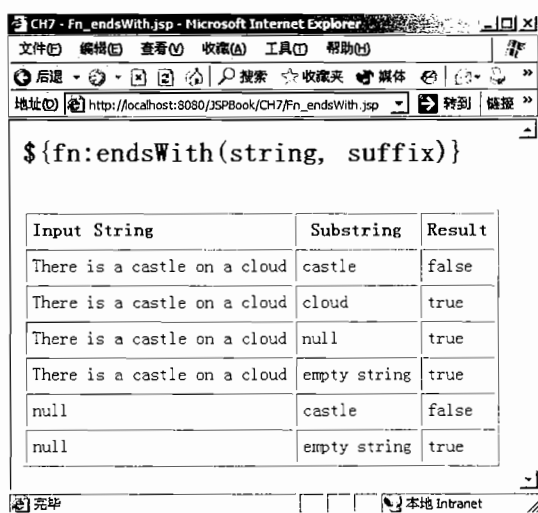


图 7-25 Fn_endsWith.jsp 的执行结果

属性

名 称	说 明	类 型
string	原输入的字符串	String
Result	转换后的字符串	String

Null 和 错误处理

- 假若 string 为 null 时, 它会以空字符串做处理

说明

`${fn:escapeXml(string)}` 函数用来转换转义字符。例如: 将 `<`、`>`、`'`、`"` 和 `&` 转换为 `<`、`>`、`'`、`"` 和 `&`。

JSP2.0 技术手册

范例

■ Fn_escapeXml.jsp

```
.....
<c:set var="s1" value="There is a castle on a cloud"/>

${fn:escapeXml(s1)}
${fn:escapeXml("<foo>body of foo</foo>")}
${fn:escapeXml(undefiend)}
${fn:escapeXml(" ")}
.....
```

执行结果如图 7-26 所示。

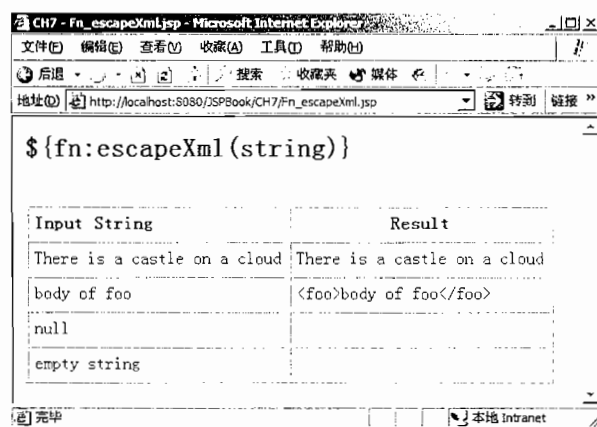


图 7-26 Fn_escapeXml.jsp 的执行结果

● <fn:indexOf>

回传某字符串到一字符串第一次吻合的位置。

语法

```
${fn:indexOf(string, substring)} => int
```

属性

名 称	说 明	类 型
string	原输入的字符串	String
substring	测试字符串	String
Result	第一次吻合的位置	int

Null 和 错误处理

同<fn:contains>

说明

`#{fn:indexOf(string, substring)}`函数用来计算 substring 在 string 中第一次相等的位置。

范例

图 Fn_indexOf.jsp

```

<!--
&ltc:set var="s1" value="There is a castle on a cloud" />

#{fn:indexOf(s1, "castle")}
#{fn:indexOf(s1, "cloud")}
#{fn:indexOf(s1, undefined)}
#{fn:indexOf(s1, "")}
#{fn:indexOf(undefined, "castle")}
#{fn:indexOf(undefined, "")}
-->

```

执行结果如图 7-27 所示。

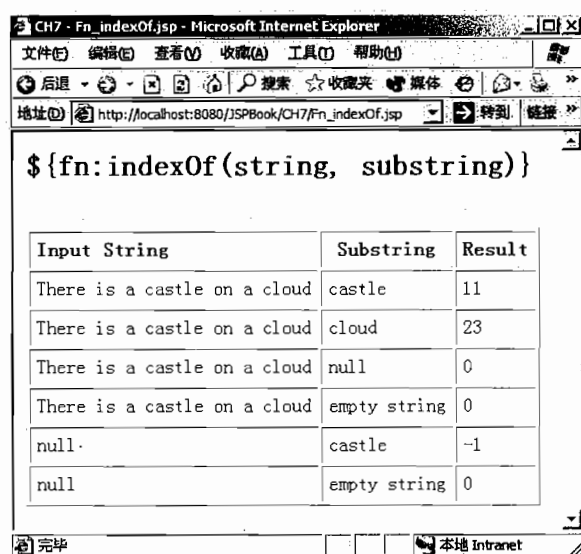


图 7-27 Fn_indexOf.jsp 的执行结果

● `<fn:split>`

将字符串分离成一字符串数组。

语法

```
#{fn:split(string, delimiters)} => string []
```

属性

名 称	说 明	类 型
string	欲被分离的字符串	String
delimiters	用来分离的字符串符号	String
Result	被分离后的字符串数组	String []

Null 和 错误处理

同<fn:contains>

说明

`${fn:split(string, delimiters)}`函数将 string 以 delimiters 元素作为分离点,然后回传分离后的字符串数组。

● <fn:join>

将数组中的全部元素以指定字符串作为连接符,回传结合后的字符串。

语法

`${fn:join(array, separator)} => string`

属性

名 称	说 明	类 型
array	被结合的数组	String []
separator	用于连接数组中的元素	String
Result	第一次吻合的位置	String

Null 和 错误处理

同<fn:contains>

说明

`${fn:join(array, separator)}`函数用 separator 将 array 中的元素连接成为一个字符串。

范例

■ Fn_split_join.jsp

```
.....
<:set var="s1" value="There is a castle on a cloud"/>
<:set var="a1" value='${fn:split(s1, " ")}'/>

${fn:join(a1, " + ")}
${fn:join(a1, " &lt;sep> ")}
```



```

${fn:join(a1, "")}
${fn:indexOf(s1, "")}
${fn:join(a1, null)}
${fn:join(null, "")}
.....

```

执行结果如图 7-28 所示。

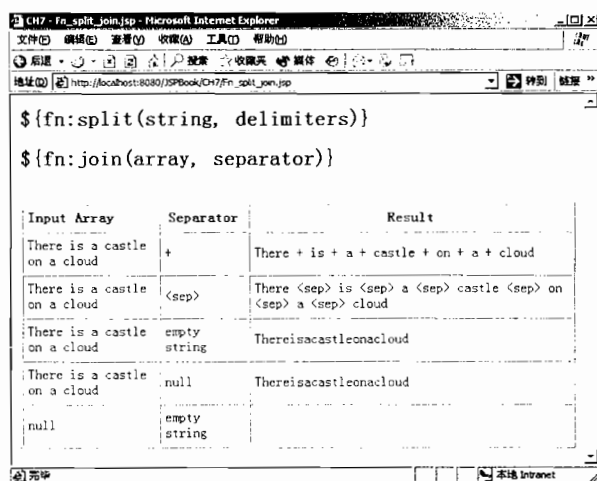


图 7-28 Fn_split_join.jsp 的执行结果

● <fn:replace>

将字符串中的某些子字符串用另一字符串取代。

语法

```
${fn:replace(inputString, beforeSubstring, afterSubstring)} => string
```

属性

名 称	说 明	类 型
inputString	原输入的字符串	String
beforeSubstring	要替换的字符串	String
afterSubstring	替换成为的字符串	String
Result	取代后的字符串	String

Null 和 错误处理

- 假若 inputString 为 null 时，它会以空字符串做处理
- 假若 beforeSubstring 为 null 时，它会以空字符串做处理
- 假若 afterSubstring 为 null 时，它会以空字符串做处理

说明

`${fn:replace(inputString, beforeSubstring, afterSubstring)}` 函数将 `inputString` 字符串中的 `beforeSubstring` 取代为 `afterSubstring`。

● `<fn:trim>`

去除字符串的前后空白。

语法

```
${fn:trim(string)} => string
```

属性

名 称	说 明	类 型
<code>string</code>	原输入的字符串	<code>String</code>
<code>Result</code>	去除前后空白后的字符串	<code>String</code>

Null 和 错误处理

- 假若 `string` 为 `null` 时，它会以空字符串做处理

说明

`${fn:trim(string)}` 函数去除 `string` 字符串前后的空白，若要去除字符串中的空白，必须使用 `${fn:replace()}` 函数。

● `<fn:substring>`

抽取字符串中的某一子字符串。

语法

```
${fn:substring(string, beginIndex, endIndex)} => string
```

属性

名 称	说 明	类 型
<code>string</code>	原输入的字符串	<code>String</code>
<code>beginIndex</code>	欲抽取的起始位置	<code>int</code>
<code>endIndex</code>	欲抽取的结尾位置	<code>int</code>
<code>Result</code>	抽取出的字符串	<code>String</code>

Null 和 错误处理

- 假若 string 为 null 时，它会以空字符串做处理
- 假若 beginIndex 大于原输入字符串的字符数，则回传空字符串
- 假若 beginIndex 小于 0 时，beginIndex 将被改为 0
- 假若 endIndex 小于 0 或大于原输入字符串时，endIndex 将被改为原输入字符串的长度
- 假若 endIndex 小于 beginIndex 时，则回传空字符串

说明

`${fn:substring(string, beginIndex, endIndex)}` 函数将抽取 string 中第 beginIndex 至 endIndex 的字符串，string 的位置由 0 开始计算。

● <fn:substringAfter>

抽取字符串中某子字符串之后的字符串。

语法

```
${fn:substringAfter(string, substring)} => string
```

属性

名 称	说 明	类 型
string	原输入的字符串	String
substring	某子字符串	String
Result	抽取出的字符串	String

Null 和 错误处理

- 假若 string 为 null 时，它会以空字符串做处理
- 假若 substring 为 null 时，它会以空字符串做处理

说明

`${fn:substringAfter(string, substring)}` 函数将抽取 string 中 substring 字符串之后的字符串。

● <fn:substringBefore>

抽取字符串中某子字符串之前的字符串。

语法

```
${fn:substringBefore(string, substring)} => string
```

属性

名 称	说 明	类 型
string	原输入的字符串	String
substring	某子字符串	String
Result	抽取出的字符串	String

Null 和 错误处理

同<fn:contains>

说明

`{fn:substringBefore(string, substring)}`函数将抽取 string 中 substring 字符串之前的字符串。

范例

■ Fn_substring.jsp

```

<!--
<:set var="zip" value="75843-5643"/>
<:set var="s1" value="There is a castle on a cloud"/>

<h2>\${fn:substring(string, beginIndex, endIndex)}</h2>
P.O. Box: ${fn:substring(zip, 6, -1)}
${fn:substring(s1, 11, 17)}
${fn:substring(s1, 12, 5)}
${fn:substring(s1, 23, -1)}
${fn:substring(s1, 23, 999)}
${fn:substring(s1, -1, -1)}
${fn:substring(s1, 99, 12)}
${fn:substring("", 2, 6)}
${fn:substring(undefined, 2, 6)}

<h2>\${fn:substringAfter(string, substring)}</h2>
P.O. Box: ${fn:substringAfter(zip, "-")}
${fn:substringAfter(s1, "There")}
${fn:substringAfter(s1, "on a")}
${fn:substringAfter(s1, "not found")}
${fn:substringAfter(s1, undefined)}
${fn:substringAfter(s1, "")}
${fn:substringAfter("", "castle")}
${fn:substringAfter(undefined, "castle")}
${fn:substringAfter(undefined, "")}

<h2>\${fn:substringBefore(string, substring)}</h2>
Zip without P.O. Box: ${fn:substringBefore(zip, "-")}
${fn:substringBefore(s1, "on a")}
${fn:substringBefore(s1, "castle")}
${fn:substringBefore(s1, undefined)}
${fn:substringBefore(s1, "")}

```

JSP2.0 技术手册

```

${fn:substringBefore("", "castle")}
${fn:substringBefore(undefiend, "castle")}
${fn:substringBefore(undefiend, "")}
.....

```

其中`${fn:substring()}`的执行结果如图 7-29 所示。

Input String	beginIndex	endIndex	Result
75843-5643	6	-1	P.O. Box: 5643
There is a castle on a cloud	11	17	castle
There is a castle on a cloud	12	5	
There is a castle on a cloud	23	-1	cloud
There is a castle on a cloud	23	999	cloud
There is a castle on a cloud	-1	-1	There is a castle on a cloud
There is a castle on a cloud	99	12	
empty string	2	6	
null	2	6	

`${fn:substringAfter(string, substring)}`

图 7-29 Fn_substring.jsp 中`${fn:substring()}`的执行结果

其中`${fn:substringAfter()}`的执行结果如图 7-30 所示。

其中`${fn:substringBefore()}`的执行结果如图 7-31 所示。

Input String	substring	Result
75843-5643	-	P.O. Box: 5643
There is a castle on a cloud	There	is a castle on a cloud
There is a castle on a cloud	on a	cloud
There is a castle on a cloud	not found	
There is a castle on a cloud	null	There is a castle on a cloud
There is a castle on a cloud	empty string	There is a castle on a cloud
empty string	castle	
null	castle	
null	empty string	

`${fn:substringAfter(string, substring)}`

图 7-30 Fn_substring.jsp 中`${fn:substringAfter()}`的执行结果

- **<fn:toLowerCase>**
转换为小写字符。

语法

`${fn:toLowerCase(string)} => string`

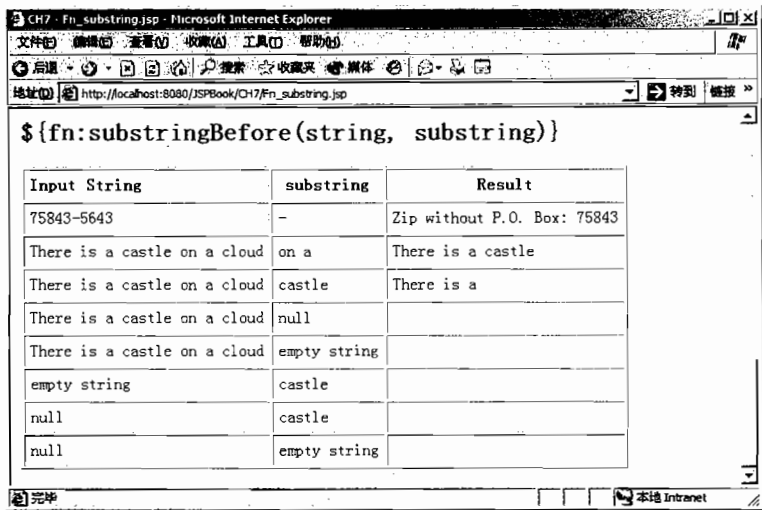


图 7-31 Fn_substring.jsp 中 `${fn:substringBefore()}` 的执行结果

属性

名 称	说 明	类 型
string	原输入的字符串	String
Result	转换为小写后的字符串	String

Null 和 错误处理

- 假若 string 为 null 时，它会以空字符串做处理并回传空字符串

说明

`${fn:toLowerCase(string)}` 函数将 string 中所有的字符都转换为小写。

- **<fn:toUpperCase>**
转换为大写字符

语法

```
${fn:toUpperCase(string)} => string
```

属性

名 称	说 明	类 型
string	原输入的字符串	String
Result	转换为大写后的字符串	String

Null 和 错误处理

- 假若 string 为 null 时，它会以空字符串做处理并回传空字符串

说明

`${fn:toUpperCase(string)}`函数将 string 中所有的字符都转换为大写。

❶ <fn:length>

回传一集合对象的数量或一字符串中的字符数。

语法

```
${fn:length(input)} => integer
```

属性

名 称	说 明	类 型
input	原输入	符合<c:forEach>中 items 属性的类型 / String
Result	input 的数量大小	int

Null 和 错误处理

- 假若 input 为 null 时，它会以空的集合对象或字符串做处理，并回传 0

说明

`${fn:length(input)}`函数将取得 input 的大小。

8

第八章

JSP 与 JavaBean

本章将为读者介绍利用 JSP 技术搭配 JavaBean 组件的架构。内容共分 4 节：

- 8-1 JavaBean 的简介
- 8-2 JSP 与 JavaBean
- 8-3 JavaBean 的范围
- 8-4 JavaBean 的移除

JSP2.0 技术手册

8-1 JavaBean 的简介

JavaBean 是一个可重复使用、且跨平台的软件组件(Software Component)，它可以在软件开发工具如：Borland JBuilder、Oracle JDeveloper 或是 Sun ONE Studio 等等里，以可视化的方式来开发。

首先，你可以将 JavaBean 视为一个黑盒子(Black Box)，虽然知道这个软件组件所具备的功能，却不知其内部是如何运作的。笔者提供给读者一个假想的情况：有一个黑盒子，只知道它上面有一个按钮，你只要按下去经过十分钟，就会掉下一张一千元钞票（哈哈，相信很多读者也想要那个黑盒子吧）。不过你看不见其内部任何的构造，而这就是 JavaBean 最主要的特性，它将许多的信息都封装了起来，用户无须知道黑盒子如何产生出钞票，只须知道要按下那个按钮，然后等十分钟，钞票就会自动产生出来。

一般而言，JavaBean 可分为：有用户接口(UI, User Interface)的 JavaBean 和没有用户接口的 JavaBean。通常 JSP 是搭配没有 UI 的 JavaBean，因此后面所提到的 JavaBean 都只是单纯处理一些事务，如：数据运算、连接数据库和数据处理，等等，至于有用户接口的 JavaBean 部分请读者自行参考 JavaBean 的相关书籍。

通常一个标准的 JavaBean 有如下几项特性：

- (1) JavaBean 是一个公开的(public)类；
- (2) JavaBean 类必须有一个无传入参数(zero-argument)的构造函数(constructor)；
- (3) 取得或设定属性(properties)时，必须使用 getXXX 方法或 setXXX 方法。

为了验证上述三点，笔者举一个 SimpleBean 的例子：

■ SimpleBean.java

```
package tw.com.javaworld.CH8;

import java.io.*;

public class SimpleBean{

    public SimpleBean() {

    }

    private String name;
    private String number;

    public void setName(String name) {
        this.name = name;
    }

    public void setNumber(String number) {
        this.number = number;
    }
}
```

JSP2.0 技术手册

```

    }

    public String getName() {
        return name;
    }

    public String getNumber() {
        return number;
    }
}

```

SimpleBean.java 是一个基本的 JavaBean 的范例，如下所示。

```

public class SimpleBean{
    .....
    .....
}

```

首先建立一个公开的 JavaBean 类，名为 SimpleBean。这具备了 JavaBean 中第一个特性。

```

public SimpleBean( ) {
    .....
    .....
}

```

设定它为 SimpleBean 类。构造函数中没有任何的传入值，这具备第二个特性。

```

private String name;
private String number;

public void setName(String name) {
    this.name = name;
}

public void setNumber(String number) {
    this.number = number;
}

public String getName() {
    return name;
}

public String getNumber() {
    return number;
}

```

SimpleBean.java 中，声明两个 String 类型的属性：name 和 number，每一个属性，都定义了二种方法：setXXX 和 getXXX，使之具备第三个特性。例如：以 name 为例，如果要取得 name 属性时，就调用 getName()这个方法；若要设定 name 属性时，就调用 setName()这个方法，这是 JavaBean 规范书中存取 JavaBean 属性的标准方法。

补充

name 和 number 在 JavaBean 中，我们通常称它们叫属性(property)，不称它们为变量。

8-2 JSP 与 JavaBean

JSP 搭配 JavaBean 来使用, 有以下优点:

- 可将 HTML 和 Java 程序分离, 为了日后维护的方便

如果把所有的程序代码(Html 和 Java)写到 JSP 网页中, 则会使整个程序代码又多又繁杂, 造成日后维护上的困难。

- 可利用 JavaBean 的优点

我们可以将常用到的程序写成 JavaBean 组件, JSP 网页只要调用 JavaBean 组件来执行我们所要的功能, 不用再重复写相同的程序, 这样一来, 也可以节省开发所需的时间。

笔者认为使用 JavaBean 最大的缺点就是麻烦, 因为制作 JavaBean 组件时, 每次都要先用 JDK 将它编译成 .class, 然后再将它放入 JSP Container 中执行, 若执行有错误, 则要做小幅度修改, 因此又要重新编译一次。不过现在我们搭配使用 Ant 来开发, 可以节省不少功夫和时间。

补充

读者可自行参阅 1-4 节的“安装 Ant 1.6”。

在搭配 JSP+JavaBean 时, 事先的规划、设计是相当重要的, 把一些使用性高且复杂的数据处理写成 JavaBean 组件。其他部分, 如果为了开发方便考虑时, 就直接写在 JSP 网页就行了。这部分就必须由程序员自行去评估、考虑了。

以下为一个 JavaBean 的范例程序, 现在就来看看如何在 JSP 中搭配使用 JavaBean。

■ SimpleBean.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<html>
<head>
  <title>CH8 - SimpleBean.jsp</title>
</head>
<body>

<h2>JSP 使用 JavaBean 的方法</h2>

<jsp:useBean id="myBean" scope="page"
class="tw.com.javaworld.CH8.SimpleBean" />
<jsp:setProperty name="myBean" property="name" value="${param.Name}" />
<jsp:setProperty name="myBean" property="number"
value="${param.Number}" />
Hi !
<font color="red"><jsp:getProperty name="myBean" property="name" /></font>
您好<br>
您的编号为:
<font color="red"><jsp:getProperty name="myBean" property="number" /></font>
```

JSP2.0 技术手册


```

/></font>
</body>
</html>

```

在执行 *SimpleBean.jsp* 时, 请自行在 URL 的最后加上“?Name=XXX&Number=XXXX”。*SimpleBean.jsp* 范例程序只是让读者了解 JSP 如何搭配 JavaBean。*SimpleBean.jsp* 的执行结果如图 8-1 所示。



图 8-1 SimpleBean.jsp 的执行结果

8-2-1 <jsp:useBean>

```

<jsp:useBean id="myBean" scope="page"
class="tw.com.javaworld.CH8.SimpleBean"/>

```

上面是 JSP 调用 JavaBean 的方法。如果要在 JSP 网页中调用 JavaBean, 就必须用到 `<jsp:useBean>` 这个标签。在 *SimpleBean.jsp* 中调用一个 JavaBean, 它的全名为 `tw.com.javaworld.CH8.SimpleBean`, 其中 `tw.com.javaworld.CH8` 是套件名称, 而 `SimpleBean` 是类名称。

现在来说明 `<jsp:useBean>` 这个标签。当我们在 JSP 网页中用到它时, 表示会产生一个 JavaBean 的实体(instance)。`<jsp:useBean>` 有五个属性: `id`、`scope`、`class`、`beanName` 和 `type`。

```

<jsp:useBean id="name" scope="page | request | session | application"
              typeSpec>

```

本体内内容

```

</jsp:useBean>

```

其中, `typeSpec` 定义如下:

```

typeSpec ::= class = "className"
            | class = "className" type = "typeName"
            | type = "typeName" class = "className"
            | beanName = "beanName" type = "typeName"
            | type = "typeName" beanName = "beanName"
            | type = "typeName"

```

● id:

在 JSP 网页中, `id` 值表示 JavaBean 的代号。由上述例子, 用 `myBean` 来代表我们使用的

JavaBean 对象。

☹ **scope:**

表示这个 JavaBean 的范围。它有四种范围：Page、Request、Session 和 Application。这和“第五章：隐含对象（Implicit Object）”介绍的四种范围一样。笔者将在 8-3 节中介绍这四种范围彼此代表的意义和差异。

☹ **class:**

表示所调用的 JavaBean 类的位置。以 *SimpleBean.jsp* 为例，如 `class = "tw.com.javaworld.CH8.SimpleBean"`。

☹ **beanName:**

`beanName` 属性代表了 Bean 的名字，通常利用 `java.beans.Beans` 类的 `instantiate()` 方法来初始化。

● **type:**

`type` 指定了 Scripting 变量定义的类型，因为 Scripting 变量定义和 `class` 中的属性一致，因此一般我们都采用默认值。

8-2-2 自省(introspection)的机制

```
<jsp:setProperty name="myBean" property="name" value="${param.Name}" />
<jsp:setProperty name="myBean" property="number" value="${param.Number}" />
```

这两行都是利用 `<jsp:setProperty>` 标签来设定 JavaBean 属性的。不过在说明 `<jsp:setProperty>` 标签之前，笔者先介绍一个很重要的机制：自省(introspection)。

所谓的自省机制是指：当服务器接收到请求时，它根据请求的参数名称，自动设定与 JavaBean 相同属性名称的值，下面举个例子说明一下：

现在笔者利用 JavaBean 自省机制，来实现 HTML+JSP+JavaBean 的架构。首先假设一个网页 *Introspection.html*，要求用户输入姓名(name)和编号(number)，当按下【传送】时，传送到 *Introspection.jsp* 执行调用 *SimpleBean*，然后显示结果。

■ *Introspection.html*

```
<html>
<head>
  <title>CH8 - Introspection.html</title>
  <meta http-equiv="Content-Type" content="text/html; charset=GB2312">
</head>
<body>

<h2>Introspection - 自省机制</h2>
<form name="form1" action="Introspection.jsp" method="post" >
  <p>姓名:
    <input type="text" name="name">
  </p>
```

JSP2.0 技术手册

```

<p>编号:
  <input type="text" name="number">
</p>
<p>
  <input type="submit" value="传送">
  <input type="reset" value="取消">
</p>
</form>
</body>
</html>

```

Introspection.html 的执行结果如图 8-2 所示。下面我们还是使用之前写好的 *SimpleBean.java* 来当做我们的 JavaBean，完整的程序代码如下：



图 8-2 *Introspection.html* 的执行结果

■ *SimpleBean.java*

```

package tw.com.javaworld.CH8;

import java.io.*;

public class SimpleBean{

    public SimpleBean() {
    }

    private String name;
    private String number;

    public void setName(String name) {
        this.name = name;
    }

    public void setNumber(String number) {
        this.number = number;
    }
}

```

```

    }

    public String getName() {
        return name;
    }

    public String getNumber() {
        return number;
    }
}

```

在 *SimpleBean.java* 中声明两个属性: *name* 和 *number*, 注意, 这两个属性名称必须要和 *Introspection.html* 中两个输入字段的名称一样。代码如下所示:

```

.....
<input type="text" name="name">
<input type="text" name="number">
.....

```

Introspection.html 有两个本文输入类型, 分别叫做 *name* 和 *number*, 和 *SimpleBean.java* 中两个属性的名称一样, 代码如下:

```

.....
private String name;
private String number;
.....

```

最后就是 *Introspection.jsp*, 它会把接收的数据显示出来。代码为:

■ *Introspection.jsp*

```

<%@ page contentType="text/html; charset=GB2312" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<html>
<head>
    <title>CH8 - Introspection.jsp</title>
</head>
<body>

<h2>Introspection - 自省机制</h2>
<fmt:requestEncoding value="GB2312" />

<jsp:useBean id="myBean" scope="page"
class="tw.com.javaworld.CH8.SimpleBean"/>
<jsp:setProperty name="myBean" property="*" />

姓名: <jsp:getProperty name="myBean" property="name"/><br>
编号: <jsp:getProperty name="myBean" property="number"/>

</body>
</html>

```

`<jsp:setProperty name="myBean" property="*" />`就是通过自省机制设定所有窗体传来

JSP2.0 技术手册

的参数, 若参数名称与 JavaBean 属性名称一样时(例如: *Introspection.html* 中有两个 text 类型 name 和 number; SimpleBean 中也有两个属性名称为 name 和 number), 就自动把参数值利用 setXXX 方法, 设定给 JavaBean 的属性。*Introspection.jsp* 的执行结果如图 8-3 所示。

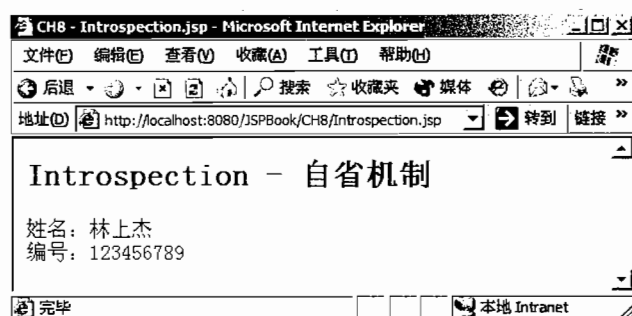


图 8-3 Introspection.jsp 的执行结果

关于<jsp:setProperty>这个标签, 在下一节有更详细的说明, 而关于<jsp:getProperty>这个标签, 则在 8-2-4 小节中再为各位读者介绍。

8-2-3 <jsp:setProperty>

一般说来, JSP 提供四种方法来设定 JavaBean 的属性值:

```
<jsp:setProperty name="myBean" property="*" />
<jsp:setProperty name="myBean" property="myProperty" />
<jsp:setProperty name="myBean" property="myProperty" param="ParamName" />
<jsp:setProperty name="myBean" property="myProperty" value="MyValue" />
```

其中 name 就是<jsp:useBean>的 id, 因此 name 要和<jsp:useBean>的 id 一样。第一种写法: 由窗体传来的参数值, 通过自省机制, 设定所有的属性值; 第二种写法也是通过自省机制, 不过只设定 myProperty 的属性值; 第三种写法, 由窗体传来的参数名称为 ParamName 的值, 传给属性名称为 myProperty; 第四种写法, 它的弹性最大, 它可以通过 value, 动态设定 JavaBean 的属性, 例如:

```
<jsp:setProperty name="my" property="number" value="${number}" />
```

为了让读者更了解这四种方法的差别, 笔者做了一连串的范例, 让读者能够了解它们的使用。HTML 网页和 JavaBean 都是利用之前的范例: *Introspection.html* 和 *SimpleBean.java*, 不过 *Introspection.jsp* 笔者只针对<jsp:setProperty>标签做修改而已, 其余部分和原来的一样。图 8-4 为 *Introspection.html* 的执行结果。



图 8-4 Introspection.html 的执行结果

第一种 <jsp:setProperty>方法

```
<jsp:setProperty name="myBean" property="*" />
```

* 表示根据窗体中所有的参数，设定 JavaBean(如: *SimpleBean.java*)的属性。在这个例子中, *Introspection.html* 有两个参数: name 和 number, 因此它设定 *SimpleBean.java* 中的 name 和 number 两个属性。这里有一个地方须要注意, 窗体的参数名称和 JavaBean 的属性名称必须要大小写一致, 才能顺利通过自省机制来完成工作, 图 8-5 为其执行结果。

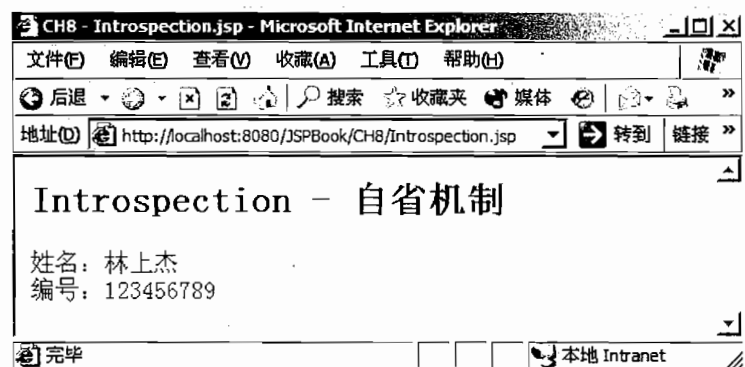


图 8-5 <jsp:setProperty name="myBean" property="*" /> 的执行结果

第二种 <jsp:setProperty>方法

如果写成如下:

```
<jsp:setProperty name="myBean" property="name" />  
<jsp:setProperty name="myBean" property="number" />
```

它的结果就会和第一种写法一样。

第二种做法是让用户能够有弹性地设定 JavaBean 的属性值。例如: *Introspection.html* 有两

个参数: name 和 number, 但是我只设定 name 到 JavaBean 中, 因此只写成:

```
<jsp:setProperty name="myBean" property="name" />
```

它只单单设定 JavaBean 的 name 属性, 而 number 属性就为 null 值。执行结果如图 8-6 所示, 姓名还是一样为林上杰, 但编号的位置却没有任何值。

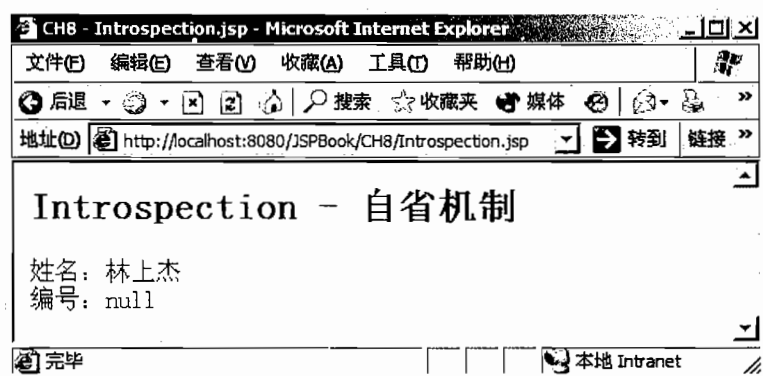


图 8-6 <jsp:setProperty name="myBean" property="name" />的执行结果

第三种 <jsp:setProperty>方法

```
<jsp:setProperty name="myBean" property="myProperty" param="ParamName" />
```

这种方法比前两种方法有弹性, 因为前两种方法有一个限制, 那就是窗体的参数名称和 JavaBean 的属性名称必须要大小写一致才能够顺利使用。如果两者名称不相同, 就不能够设定 JavaBean 的属性值, 此时可以使用第三种方式来达成。

param 即为窗体的参数名称, 例如: 当我们的窗体参数为: myName 和 myNumber, 而 SimpleBean.java 的属性名称为: name 和 number, 此时可以写成如下形式:

```
<jsp:setProperty name="myBean" property="name" param="myName" />
<jsp:setProperty name="myBean" property="number" param="myNumber" />
```

就可以顺利达成我们所要的结果。

不过笔者在此举一个例子, 在不修改 Introspection.html 的情况下, 将 Introspection.jsp 部分的程序代码改为如下形式:

```
<jsp:setProperty name="myBean" property="name" param="number" />
<jsp:setProperty name="myBean" property="number" param="name" />
```

这段程序主要将窗体 number 的值设定给 JavaBean 的 name 属性; 窗体 name 的值设定给 JavaBean 的 number 属性, 简单地说, 就是两者做一个对调。执行结果如图 8-7 所示。

第四种 <jsp:setProperty>方法

```
<jsp:setProperty name="myBean" property="myProperty" value="MyValue" />
```

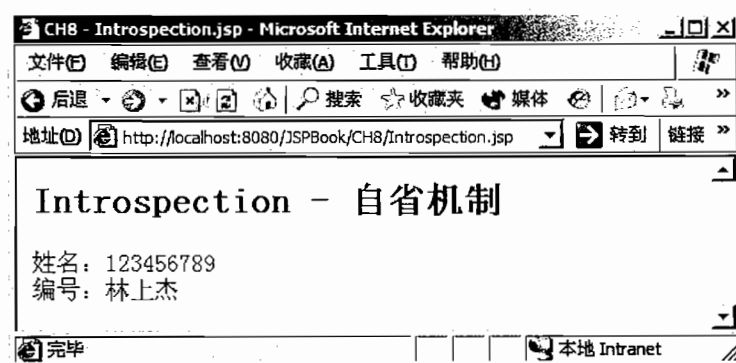


图 8-7 `<jsp:setProperty name="myBean" property="myProperty" param="ParamName" />` 的执行结果

这种方法比前三种方法更有弹性, 因为前三种方法都是将窗体中参数的值设定到 JavaBean 的属性之中, 而这种方法可以让用户动态设定 JavaBean 属性的值, 代码如下:

```
<jsp:setProperty name="myBean" property="name" value="mike" />
<jsp:setProperty name="myBean" property="number" value="12345" />
<jsp:setProperty name="myBean" property="price" value="1,200" />
```

除了可以设定一个静态的值之外, 它还可以设定一个动态产生出的值给 JavaBean 的属性, 例如:

例 1:

```
<jsp:setProperty name="myBean" property="name" value="${param.name}" />
```

例 2:

```
<jsp:setProperty name="myBean" property="number" value="${num}" />
```

第一个例子的写法, 其实也就等于下面的写法:

```
<jsp:setProperty name="myBean" property="name" param="name" />
```

第二个例子是直接指定一个变量 num 值给这个 number 属性, 当然你在 JSP 网页中要事先声明这个变量。最后当然也要修改 *Introspection.jsp*, 举个实际的例子如下:

```
<jsp:setProperty name="myBean" property="name" value="mike" />
<jsp:setProperty name="myBean" property="number" value="1234" />
```

它们的执行结果如图 8-8 所示。

相信读者看完上述内容之后, 应该知道如何利用 `<jsp:setProperty>` 标签将窗体的数据传入 JavaBean 中。不过除了上述的功能之外, 还有一项便利的功能, 那就是数据类型自动转换功能。下面举个例子说明:

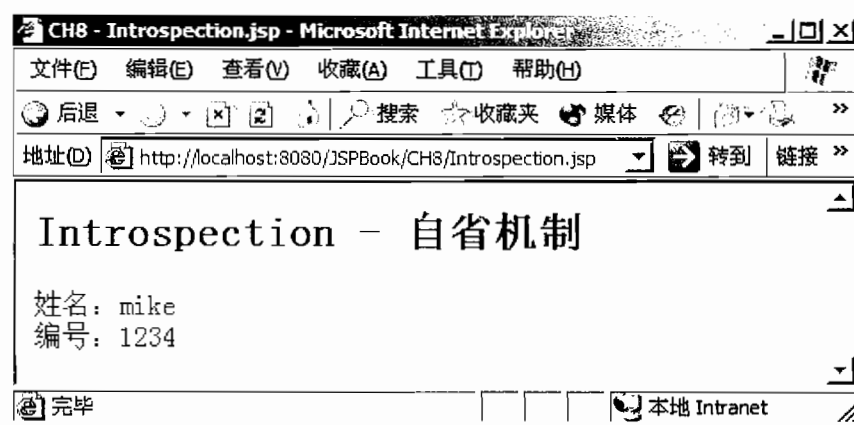


图 8-8 `<jsp:setProperty name="myBean" property="myProperty" value="MyValue" />` 的执行结果

如果你要处理的数据本身是一个 String 类型，代码如下所示：

```
<jsp:useBean id="myBean" class="myPackage.myBean" />
<jsp:setProperty name="myBean" property="name" value="${param.name}" />
```

注意

JSP 2.0 之后，使用 bean 时一定要加 package name。

不过，当处理的类型不是 String 时，必须要将数据类型做转换，如下面这个例子：

JavaBean 的属性 age 为 int 类型时，因为从窗体传来的数据类型都为 String，因此必须先将它转为 int 类型，再传入 JavaBean 做处理。

```
<jsp:useBean id="myBean" class="myPackage.myBean" />
<%
    int age = 0;
    try {
        age = Integer.parseInt(request.getParameter("age"));
    }
    catch (NumberFormatException ex) {
        age = AGE_UNKNOWN;
    }
%>
<jsp:setProperty name="myBean" property="age" value="${age}" />
```

假如你的窗体里面有大量的 int、double 类型的变量，像这样的转换工作就显得非常繁琐，不过幸好 JSP 网页能提供解决这类问题的数据类型自动转换功能。

这个功能可以把一个特定请求的参数类型与 JavaBean 中的属性类型自动关联，它主要方便在于：不须手动进行参数类型转换，而是由系统自动根据 JavaBean 属性的定义进行类型转换。因此有了这项功能时，上面那个程序代码就可以改写为：

JSP2.0 技术手册

```
<jsp:useBean id="myBean" class="myPackage.myBean" />
<jsp:setProperty name="myBean" property="age" param="age" />
```

两相比较，现在是不是简便多了。

补充

不过这里也有要小心的地方，千万不能写为下列的形式：

```
<jsp:useBean id="myBean" class="myPackage.myBean" />
<jsp:setProperty name="myBean" property="age" value=
    "<%= request.getParameter(\"age\") %>" />
```

如果这样写，那会产生以下的错误信息：

Attribute age has no value

因为 JavaBean 的 age 属性是 int 类型的，而使用 `<%= %>` 回传的数据类型一定是 String 类型，因此会产生错误，这是读者在使用时必须小心的地方。但是如果使用 EL 的写法，就没有这个问题了：

```
<jsp:useBean id="myBean" class="myPackage.myBean" />
<jsp:setProperty name="myBean" property="age" value="${age}" />
```

那是因为 EL 会自动帮我们转换正确的类型。

系统提供的数据类型自动转换只能作用于简单类型，表 8-1 是针对不同的属性类型所进行的转换。

表 8-1 属性类型所进行的转换

属性类型	转换方法
boolean	java.lang.Boolean.valueOf(String param)
Boolean	java.lang.Boolean.valueOf(String param)
byte	java.lang.Byte.valueOf(String param)
Byte	java.lang.Byte.valueOf(String param)
char	java.lang.Character.valueOf(String param)
Character	java.lang.Character.valueOf(String param)
double	java.lang.Double.valueOf(String param)
Double	java.lang.Double.valueOf(String param)
int	java.lang.Integer.valueOf(String param)
Integer	java.lang.Integer.valueOf(String param)
float	java.lang.Float.valueOf(String param)
Float	java.lang.Float.valueOf(String param)
long	java.lang.Long.valueOf(String param)
Long	java.lang.Long.valueOf(String param)

需要注意的是：这些转换并不包含参数中有非法字符的情况，也就是说，照样有可能产生 `NumberFormatException`。你可以加强客户端的数据校对，或者利用 `errorPage` 进行错误处理。

注意

当请求中根本没有你指定的参数时，此时自动转换功能不会把 `getParameter` 取得的 `null` 送入 `JavaBean` 中，而是不进行任何动作，于是无法及时将输入不完整的信息反馈(feedback)给服务器。因此你最好在 `JavaBean` 中赋予属性默认值，并且在最后的数据处理阶段做判断。总之，数据校对的工作并不会因此变得轻松。

8-2-4 <jsp:getProperty>

若要取得 `JavaBean` 中的属性值，我们必须使用 `<jsp:getProperty>` 这个标签。`<jsp:getProperty>` 标签不像 `<jsp:setProperty>` 标签有那么多种使用方法，而只有一种，如下所示：

```
<jsp:setProperty name="myBean" property="myProperty" />
```

因此使用的方法很简单，例如：*Introspection.jsp* 显示 `name` 和 `number` 属性的值，我们只需要下列程序：

```
<jsp:useBean id="myBean" scope="page"
  class="tw.com.javaworld.CH8.SimpleBean"/>

<jsp:getProperty name="myBean" property="name"/>
<jsp:getProperty name="myBean" property="number"/>
```

其中 `name` 就是 `<jsp:useBean>` 的 `id`，因此 `name` 要和 `<jsp:useBean>` 的 `id` 一样，而 `property` 就是想要取得的属性值。

`<jsp:getProperty>` 和 `<jsp:setProperty>` 一样也有数据类型自动转换功能，不过 `<jsp:getProperty>` 是将各类型的属性类型皆转为 `String` 的类型，然后显示至网页上，表 8-2 列出了属性类型所进行的转换的列表：

表 8-2 属性类型所进行的转换的列表

属性类型	转换方法
boolean	<code>java.lang.Boolean.toString(boolean)</code>
byte	<code>java.lang.Byte.toString(byte)</code>
char	<code>java.lang.Character.toString(char)</code>
double	<code>java.lang.Double.toString(double)</code>
int	<code>java.lang.Integer.toString(int)</code>
float	<code>java.lang.Float.toString(float)</code>
long	<code>java.lang.Long.toString(long)</code>

8-3 JavaBean 的范围

如果读者还记得第五章的内容, 应该知道 JSP 网页有四种范围: Page、Request、Session 和 Application。使用 JavaBean 时, 我们同样也可以设定 JavaBean 的范围, 它和第五章所谈论到的范围, 除了名称一样之外, 意义也是一样的。

本节除了有文字的叙述之外, 笔者还举了几个范例程序来说明这四种范围的差异, 相信对读者会有很大的帮助。

在<jsp:useBean>标签里有一属性 scope, 它用来设定 JavaBean 的范围, 它的值只能为 page、request、session 和 application, 不可为其他的值, 如: null 或是空白, 等等。如果读者想要设定你的 JavaBean 范围为 Page 时, 可写成如下形式:

```
<jsp:useBean id="myBean" scope="page"
  class="tw.com.javaworld.CH8.SimpleBean"/>
```

表示将 JavaBean 放在 pageContext 对象中, 且只能在本页面内部使用; request 表示放在 request 对象中, 在当前 request 的处理期间都能够取用; session 表示放在 session 对象中, 只要在 session 的期间都能够取用; application 表示放在 application 对象中, 只要在服务器运作时, 都能够取用它。使用不同的 scope 属性值, 能在不同的范围内共享 JavaBean。

8-3-1 Page 范围的 JavaBean

当使用<jsp:useBean>标签时, 假若你没有指定 JavaBean 的 scope 时, 则 Container 范围的默认值为 Page, 因此下面两行程序的意思是相同的:

```
<jsp:useBean id = "myBean" class = "myPackage.myBean" />
<jsp:useBean id = "myBean" class = "myBean" scope = "page" />
```

当 JavaBean 的范围设为 Page 时, 表示这个 JavaBean 的生命周期只在一个页面里, 你只能在同一网页中去存取、利用它。假若程序涉及其他页面时, 此时 JSP Container 会自动释放其内存, 结束其生命周期。

现在笔者举一个计数器的范例程序, 它包含一个 Counter 的 JavaBean 和 Counter.jsp, 用来显示目前的参观次数。

■ Counter.java

```
package tw.com.javaworld.CH8;
```

```
public class Counter {
```

```
    public Counter() {
```

```
    }
    private int count = 0;
```

JSP2.0 技术手册


```
public int getCount( ) {  
    count ++ ;  
    return count;  
}  
public void setCount(int newCount) {  
    count = newCount;  
}  
}
```

Counter.java 中, 设定一 `int` 类型的 `count` 变量, 用来显示参观次数, 默认值为 0。当 `getCount()` 方法被调用时, 就会将 `count` 值加 1。

■ *Counter.jsp*

```
<%@ page contentType="text/html; charset=GB2312" %>  
  
<html>  
<head>  
    <title>CH8 - Counter.jsp</title>  
</head>  
<body>  
  
    <h2>范围为 Page 的 JavaBean 范例程序 - Counter</h2>  
  
    <jsp:useBean id="myBean" scope="page"  
class="tw.com.javaworld.CH8.Counter"/>  
    <p>您已访问<font color="red">  
    <jsp:getProperty name="myBean" property="count"/>  
    </font>次</p>  
    <p>欢迎再次访问</p>  
  
</body>  
</html>
```

Counter.jsp 只是简单地调用 JavaBean, 并且显示 Counter 的 `count` 属性值。*Counter.jsp* 的执行结果如图 8-9 所示。

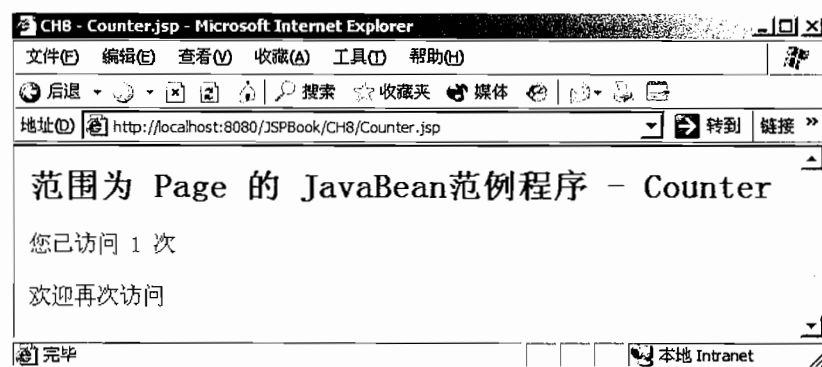


图 8-9 Counter.jsp 的执行结果

当用户在执行 *Counter.jsp* 时,主要观察 reload(刷新 F5)后的变化,你会发现,重复执行时,它的值永远保持为 1,不会有递增的结果产生。这是因为当用户重新按浏览器的 F5(刷新)后,Container 会将之前 Page 范围的 JavaBean 删除掉,然后再重新产生一个新的 Counter JavaBean,因此它的值永远会保持为 1。不过,如果范围为 session 或是 application 时,则会产生不同的结果,因此 session 和 application 可用相同的例子来说明,主要让读者能够相互比较,了解其差异性。

假若你使用<jsp:include>和<jsp:forward>标签时,所有范围为 Page 的 JavaBean 在新的或是包含进来的网页中,皆没有办法被存取。假若希望能够存取此 JavaBean,它的范围就必须设为 Request,这就是下节中所要谈论的内容。

8-3-2 Request 范围的 JavaBean

Request 范围的 JavaBean,它的生命周期和 request 对象有着不可分的关系,它的存取范围除了整个网页之外,当使用<jsp:include>或<jsp:forward>标签时,被 include 或是 forward 的网页,亦可以存取到原来网页所产生的 JavaBean。

现在笔者修改前面的 *Counter.jsp* 范例,并且改名为 *Counter1.jsp*。*Counter1.jsp* 中除了范围改为 Request 之外,并且还会 include 一个新的网页 *Hello.jsp*,因为只有加上<jsp:include>标签,才能让读者看到 Request 范围和 Page 范围的 JavaBean 有什么不同之处。

■ Counter1.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>

<html>
<head>
  <title>CH8 - Counter1.jsp</title>
</head>
<body>

  <h2>范围为 Request 的 JavaBean 范例程序 - Counter1</h2>

  <jsp:useBean id="myBean" scope="request"
    class="tw.com.javaworld.CH8.Counter"/>
  <p>您已访问<font color="red">
  <jsp:getProperty name="myBean" property="count"/>
  </font>次</p>
  <p>欢迎再次访问</p>

  <jsp:include page="Hello.jsp" flush="true"/>

</body>
</html>
```

■ Hello.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>
```

JSP2.0 技术手册

Hello...您好感谢您<jsp:getProperty name="myBean" property="count"/>次的光临

Hello.jsp 最重要的工作就是显示出 *Counter* JavaBean 的 *count* 属性值, *Hello.jsp* 能够直接存取由 *Counter1.jsp* 产生的 JavaBean, 主要是因为之前在产生 *Counter* JavaBean 时, 设定其范围为 *Request*; 假若为 *Page* 时, 在执行时会产生错误信息。图 8-10 为 *Counter1.jsp* 的执行结果。



图 8-10 Counter1.jsp 的执行结果

从图 8-10 可以看到 *Hello.jsp* 已经加入到 *Counter1.jsp* 之中, 而 *Hello.jsp* 中的参观次数为 2, 这有很重要的意义: 因为程序如果调用到 *Counter.java* 的 *getCount()* 方法时, *count* 数会先自动加 1, 然后再显示出来, 而在 *Counter1.jsp* 之中调用一次 *getCount()* 方法, 然后在 *Hello.jsp* 中又调用到同一个 JavaBean 的 *getCount()* 方法, 因此其结果各为 1 和 2。

8-3-3 Session 范围的 JavaBean

现在, 读者只须知道一个概念: 当有一个用户来我们的网站时, 可以通过 *session* 的机制来对用户进行追踪, 例如: 购物车的功能都用这个原理来做, 只有这样, 才不会把我想买的物品放置到别人的购物车, 或是别人所买的物品错放到我的购物车上。

现在把之前 *Counter.jsp* 的程序再做小幅度的修改, 只把 JavaBean 的范围改为 *Session*, 并且重新命名为 *Counter2.jsp*。

■ Counter2.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>

<html>
<head>
  <title>CH8 - Counter2.jsp</title>
</head>
<body>

  <h2>范围为 Session 的 JavaBean 范例程序 - Counter2</h2>
```

```
<jsp:useBean id="myBean" scope="session"
class= "tw.com.javaworld.CH8.Counter"/>
<p>您已访问<font color="red">
<jsp:getProperty name="myBean" property="count"/>
</font>次</p>
<p>欢迎再次访问</p>

</body>
</html>
```

Counter2.jsp 和 *Counter.jsp* 的执行结果看似没有什么不同,但是如果读者在按下浏览器的 F5 刷新后,会发现 *Counter2.jsp* 中的次数会随着 reload 的次数而逐一递增。不过假若你又另外打开一个浏览器时,再重新执行 *Counter2.jsp*,会发现数字又会从 1 开始往上递增。

综合以上结果可以得到一个结论,那就是服务器认得你,因为如果它不认得你,你的计数器不会逐一递增,并且每当另开一个浏览器窗口时,服务器会再重新产生一个新的 Counter JavaBean 来做计数的工作。

注意

假若你是直接从浏览器的 文件 | 新增 | 窗口 来产生新的浏览器窗口,两者的 session ID 一样,所以会继续累加,并不会从 1 开始计算。

图 8-11 为 *Counter2.jsp* 的执行结果。

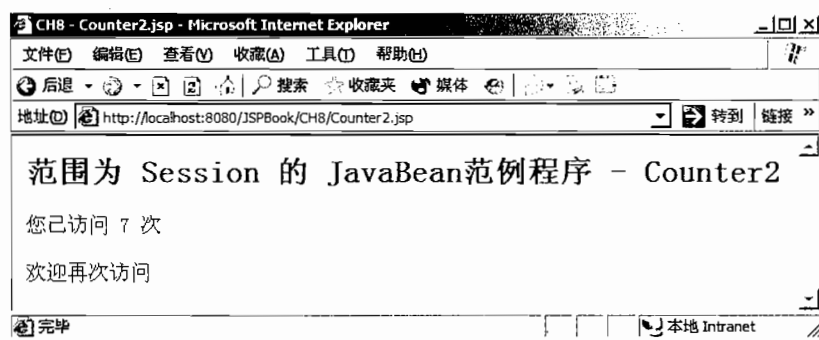


图 8-11 Counter2.jsp 的执行结果

8-3-4 Application 范围的 JavaBean

最后一个范围是 Application,它是生命周期最长久的,也可以说只要服务器不重新启动开机,它就永远存在于服务器的内存之中。因此范围设为 Application 的 JavaBean,可以在任何的页面、情况下存取到它的属性值,因此使用起来相当的便利。不过要小心地使用它,这样才不会带来一些无谓的麻烦,例如:占用过量的内存等等。

按照之前的惯例,我们还是只修改 *Counter.jsp* 的范围为 Application,改为 *Counter3.jsp*。

■ Counter3.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>

<html>
<head>
  <title>CH8 - Counter3.jsp</title>
</head>
<body>

  <h2>范围为 Application 的 JavaBean 范例程序 - Counter3</h2>

  <jsp:useBean id="myBean" scope="application"
    class="tw.com.javaworld.CH8.Counter" />
  <p>您已访问<font color="red">
  <jsp:getProperty name="myBean" property="count" />
  </font>次</p>
  <p>欢迎再次访问</p>

</body>
</html>
```

Counter3.jsp 和 Counter2.jsp 的执行结果一样, 在按下浏览器的 F5 刷新后, 计数器就会开始递增。不过两者最大的差异就在于, 假若你又另外打开一个浏览器执行 Counter3.jsp, 你会发现它不像 Counter2.jsp 那样会重新开始计算, 而是会连续下去计数。这是因为当 Counter3.jsp 产生 Application 范围的 JavaBean 时, 每次都执行第一次产生的 Counter JavaBean。图 8-12 为 Counter3.jsp 的执行结果。

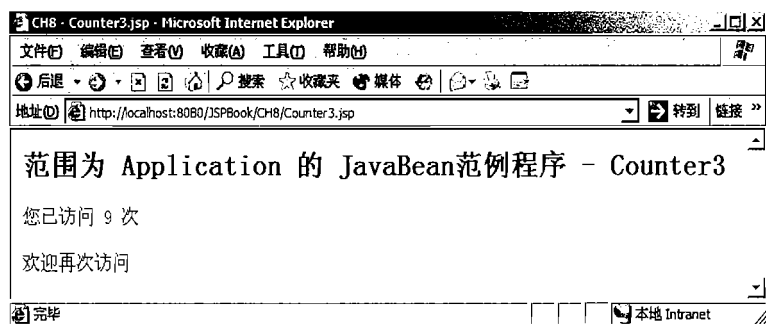


图 8-12 Counter3.jsp 的执行结果

8-4 JavaBean 的移除

这里将讨论当 JavaBean 使用完后, 想要从 Container 释放其内存时, 移除 JavaBean 的方法。

通常一个 JavaBean 的产生, 会根据它的范围来决定它的生命周期, 而它的存在会占用服务器的内存空间, 因此为了能够让服务器的性能维持一定的水准, 通常我们会先估评 JavaBean 是

否经常使用，若不是经常使用，当 JavaBean 的工作完成后，就应该顺手将 JavaBean 从 Container 中移除掉，来保持服务器的最佳性能。

从 Container 中移除 JavaBean 是一件很简单的事情，因为当 JavaBean 的 scope 属性为 Page 时，就表示 JavaBean 放在 pageContext 对象中，因此要移除时，只需用下列程序：

```
pageContext.removeAttribute(String name) ;
```

或是使用 EL 的语法：

```
<c:remove var="name" scope="page" />
```

name 即为 JavaBean 的 ID，同样地，Request 表示放在 request 对象中，用下列程序，即可移除 Request 范围的 JavaBean：

```
request.removeAttribute(String name) ;
```

或是使用 EL 的语法：

```
<c:remove var="name" scope="request" />
```

Session 表示放在 session 对象中，而 Application 表示放在 application 对象中，因此要移除时，可用：

```
session.removeAttribute(String name) ;  
application.removeAttribute(String name) ;
```

或是使用 EL 的语法来完成：

```
<c:remove var="name" scope="session" />  
<c:remove var="name" scope="application" />
```

笔者把它们整理成一个表格，如表 8-3。

表 8-3 移除不同范围的 JavaBean 的方法

范 围	移除的方法
page	pageContext.removeAttribute(String name) 或 <c:remove var="name" scope="page" />
request	request.removeAttribute(String name) 或 <c:remove var="name" scope="request" />
session	session.removeAttribute(String name) 或 <c:remove var="name" scope="session" />
application	application.removeAttribute(String name) 或 <c:remove var="name" scope="application" />

在这里我再利用前面的 Counter1.jsp 来当做范例，对 Counter1.jsp 做了小幅度的修改，完整程序代码如下：

■ *Remove.jsp*

```
<%@ page contentType="text/html; charset=GB2312" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<html>
<head>
  <title>CH8 - Remove.jsp</title>
</head>
<body>

  <h2>删除 JavaBean</h2>

  <jsp:useBean id="myBean" scope="request"
class="tw.com.javaworld.CH8.Counter" />
  <p>您已访问<font color="red">
<jsp:getProperty name="myBean" property="count" />
</font>次</p>
  <p>欢迎再次访问</p>

  <c:remove var="myBean" scope="request" />
  <jsp:include page="Hello.jsp" flush="true" />

</body>
</html>
```

Remove.jsp 和 *Counter1.jsp* 惟一不同的地方在于：我在 include *Hello.jsp* 网页之前，已经把 Request 范围的 JavaBean 删除掉，而且 `<jsp:include>` 的 `flush` 设为 `true`，所以 *Remove.jsp* 的执行结果如图 8-13 所示。



图 8-13 *Remove.jsp* 的执行结果

假若 `<jsp:include>` 的 `flush` 设为 `false` 时，则执行结果如图 8-14 所示：

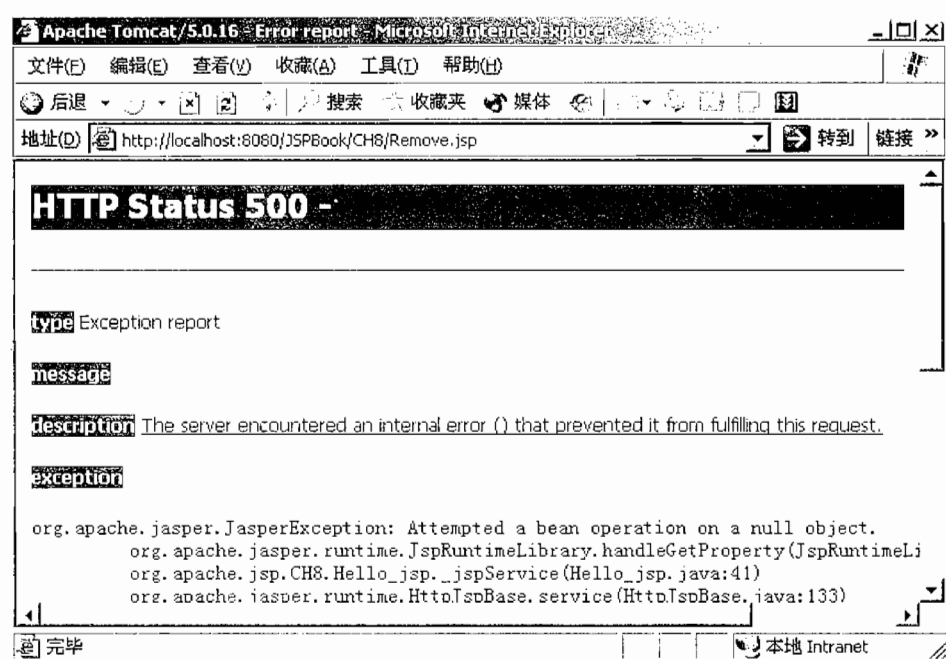


图 8-14 <jsp:include>的 flush=false 时, Remove.jsp 的执行结果

9

第九章

网页窗体的处理

各位读者当您在浏览网页时，时常可以看到让来访用户留下意见、建议，提供双向互动的功能，这些供输入数据的字段都统称为窗体(Form)。本章主要讨论如何利用 JSTL、EL 来取得用户所输入的数据，然后再经过运算处理，最后把结果传回到前端。本章分六节来介绍。

- 9-1 HTML 窗体如何传送数据
- 9-2 窗体中常见的输入类型
- 9-3 JSP 处理窗体
- 9-4 文件上传——Oreilly 上传组件
- 9-5 jspSmartUpload——上传和下载
- 9-6 本文区输入类型 (Textarea)

JSP2.0 技术手册

9-1 HTML 窗体如何传送数据

HTML 窗体提供许多的用户接口控制, 让用户能够通过网页, 输入数据。每个控件都有其名称和值, 而这些控件的内容都必须放在<FORM>...</FORM>标签之中, 当用户确定好窗体后, 控件的名称和值就会被当做一字符串送到指定的 URL 中。

Name1=Value1&Name2=Value2 ... NameN=ValueN

一般来说, 目前最常使用的传送方式有两种, 第一种是 GET, 它将这些数据加在指定好的 URL 之后, 中间用问号连接, 然后传送至指定的程序做处理, 如下所示:

http://specifiedURL? Name1=Value1&Name2=Value2

第二种传送方式为 POST。这里控件的名称、值和 URL 是分开传送的, 所以它的表现方式也就不会像 GET 一样, 你看到的也只是一段 URL 的字符串, 并无控件的字符串出现。

http://specifiedURL

采用 POST 的情形通常会有以下三种:

- (1) 为了传送较大的数据。因为使用 GET 来传递数据时, 有数据量的限制(255 个字符数)。
- (2) 执行上传文件时。因为文件大小通常都会超过 255 个字符数或文件可能是二进制文件。
- (3) 防止重要数据直接在浏览器上显示。例如: 传递用户名称、密码时, 若使用 GET 来传递, 名称和密码皆会显示在浏览器的 URL 上。

9-2 窗体中常见的输入类型

本节主要介绍窗体中常用到的输入类型, 这部分笔者将会使用 EL 和 JSTL 的语法来处理窗体。不过这里有两个地方必须要注意:

- (1) 上传文件的处理;
- (2) Textarea 标签的处理。

笔者将针对这两个主题多加着墨。接下来将介绍以下常见的输入类型:

- (1) 本文输入类型 (Text);
- (2) 密码输入类型 (Password);
- (3) 选项按钮输入类型 (Radio);
- (4) 复选框输入类型 (Check);
- (5) 选择标签 (Select);
- (6) 文件输入类型 (File);
- (7) 本文区输入类型 (Textarea)。

JSP2.0 技术手册

9-2-1 本文、密码输入类型

■ 本文输入类型

本文输入类型是最常使用到的，它的字段是单行输入，并且可以设定用户最多可以在字段中输入特定个数的字符。本文字段的属性如表 9-1：

```
<INPUT TYPE = "TEXT" >
```

表 9-1 本文输入字段的属性

属性名称	描 述
NAME	本文字段的名称
MAXLENGTH	本文字段最多允许输入的字符个数
SIZE	本文字段的宽度
VALUE	本文字段的默认值

■ 密码输入类型

密码输入类型最主要的功能就是遮掩用户所输入之数据，但是，也只有遮掩的功能，并没有将用户所输入的数据做任何的编码。密码字段的属性如表 9-2 所示。

```
<INPUT TYPE = "PASSWORD" >
```

表 9-2 密码输入字段的属性

属性名称	描 述
NAME	密码字段的名称
MAXLENGTH	密码字段最多允许输入之字符个数
SIZE	密码字段的宽度
VALUE	密码字段的默认值

9-2-2 选项按钮、复选框、选择标签

■ 选项按钮

选项按钮提供一组选项供用户做选择，不过只能**单选**。假若选项按钮的名称一样时，表示这些具有相同名称的选项按钮被视为同一群组，但还是只能选择一个。选项按钮的属性如表 9-3 所示。

```
<INPUT TYPE="RADIO" >
```

表 9-3 选项按钮字段的属性

属性名称	描 述
CHECKED	表示此选项按钮为默认选取
NAME	选项按钮的名称
VALUE	选项按钮的值，若选项按钮被选取时，此值将被传送至服务器做处理

■ 复选框

复选框提供一组选项供用户做选择，不过这个选项能够**多重选择**，这是和选项按钮最大不同的地方。复选框的属性如表 9-4 所示。

```
<INPUT TYPE="CHECKBOX">
```

表 9-4 复选框字段的属性

属性名称	描 述
CHECKED	表示此复选框为默认选取
NAME	复选框的名称
VALUE	复选框的值，若复选框被选取时，此值将被传送至服务器做处理

■ 选择标签

选择标签提供一个下拉式的菜单，供用户选取。它把复选框、选项按钮的功能结合起来，因此它能够个别提供单选或是多重选择，而且选择标签所占用的空间较小，因此广为使用。选择标签的属性如表 9-5 所示。

```
<SELECT NAME="name" >  
  <OPTION VALUE="value">选项</OPTION>  
</SELECT>
```

表 9-5 选择标签的属性

属性名称	描 述
MULTIPLE	此选项表示此选择标签为多重选择
SIZE	选择标签字段的大小
NAME	选择标签的名称
OPTION VALUE	选择标签的值，若选择标签被选取时，此值将被传送至服务器做处理

9-3 JSP 处理窗体

本节将综合上述五个输入类型来设计一个窗体，文件名为 *Form.html*，窗体将会把用户输入的数据传给 JSP 网页名为 *Form.jsp* 做处理。*Form.html* 窗体的内容包含姓名、密码、性别、年龄和兴趣。姓名的字段类型是本文；密码的字段类型是密码；性别的字段类型是选项按钮；年龄的字段类型是选择标签，最后兴趣的字段是复选框。

■ *Form.html*

```
<html>
<head>
  <title>CH9 - Form.html</title>
<meta http-equiv="Content-Type" content="text/html; charset=GB2312">
</head>
<body>

<form name="Example" method="post" action="Form.jsp">
<p> 姓名: <input type="text" name="Name" size="15" maxlength="15"></p>
<p>密码: <input type="password" name="Password" size="15"
      maxlength="15"></p>
<p>性别: <input type="radio" name="Sex" value="Male" checked>男
      <input type="radio" name="Sex" value="Female">女</p>
<p>年龄:
  <select name="Old">
    <option value="10">10 ~ 20</option>
    <option value="20" selected>21 ~ 30</option>
    <option value="30">31 ~ 40</option>
    <option value="40">41 ~ 65</option>
  </select>
</p>
<p> 兴趣:
  <input type="checkbox" name="Habit" value="Read">
    看书
  <input type="checkbox" name="Habit" value="Game">
    电玩
  <input type="checkbox" name="Habit" value="Travel">
    旅游
  <input type="checkbox" name="Habit" value="Music">
    听音乐
  <input type="checkbox" name="Habit" value="Tv">
    看电视</p>
<p>
<input type="submit" value="传送">
  <input type="reset" value="清除">
</p>
</form>

</body>
</html>
```

JSP2.0 技术手册

Form.html 的执行结果如图 9-1，接下来我们把窗体的数据都填好，然后按下【传送】，就会将数据传至 *Form.jsp* 做处理。

接下来看我们的 JSP Page 要如何来接收由 *Form.html* 所传来的值。

图 9-1 *Form.html* 的执行结果，并且输入数据

■ *Form.jsp*

```
<%@ page contentType="text/html; charset=GB2312" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>

<html>
<head>
  <title>CH9 - Form.jsp</title>
</head>
<body>

<h2>使用 EL、JSTL 处理表单数据</h2>
<fmt:requestEncoding value="GB2312" />

姓名: <c:out value="${param.Name}" default="Nothing" /> <br>
密码: <c:out value="${param.Password}" default="Nothing" /> <br>
性别: <c:if test="${param.Sex == 'Male'}">男</c:if>
      <c:if test="${param.Sex == 'Female'}">女</c:if>
年龄: <c:choose>
      <c:when test="${param.Old == 10}">10 ~ 20</c:when>
      <c:when test="${param.Old == 20}">21 ~ 30</c:when>
      <c:when test="${param.Old == 30}">31 ~ 40</c:when>
      <c:otherwise>41 ~ 65</c:otherwise>
    </c:choose>
```

```

兴趣: <c:forEach items="${paramValues.Habit}" var="habit">
    <c:choose>
        <c:when test="${habit == 'Read'}"><li>看书</li></c:when>
        <c:when test="${habit == 'Game'}"><li>电玩</li></c:when>
        <c:when test="${habit == 'Travel'}"><li>旅游</li></c:when>
        <c:when test="${habit == 'Music'}"><li>听音乐</li></c:when>
        <c:when test="${habit == 'Tv'}"><li>看电视</li></c:when>
    </c:choose>
</c:forEach>
</body>
</html>

```

JSP 2.0 之前的做法都是使用 `request.getParameter()` 或 `request.getParameterValues()` 来接收数据。JSP 2.0 之后, 因为加入 EL 的语法, 所以我们可以更方便来处理表单的数据。假若读者对于 EL、JSTL 语法有不熟的地方, 请自行参考第六章和第七章。图 9-2 是 *Form.jsp* 的执行结果。

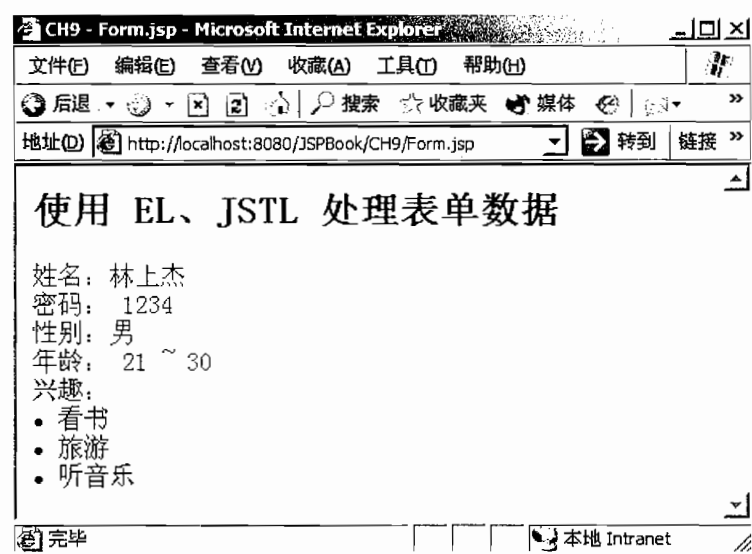


图 9-2 Form.jsp 的执行结果

9-4 文件上传——Oreilly 上传组件

文件输入类型允许用户将自己计算机内的文件上传至服务器端。文件输入类型的属性如表 9-6 所示。

```

<FORM ENCTYPE="multipart/form-data">
<INPUT TYPE="File">

```

JSP2.0 技术手册

表 9-6 文件输入类型的属性

属性名称	描 述
ACCEPT	设定用户能够上传文件之 MIME 类型
SIZE	文件输入字段的大小
NAME	文件输入字段的名称
VALUE	默认的文件名称
MAXLENGTH	文件名称最大长度限制

程序员在使用到这个输入类型时，本以为只要利用 `request.getParameter()` 的方式就能够取得文件输入类型的数据，后来才发现到，处理文件类型的方式不像之前那样的简单。为什么会在处理上有差异？如果有差异应该要如何解决？接下来就……为各位读者解答。

9-4-1 文件上传问题的原由

一般的输入类型 (例如: `text`、`password`、`radio`、`checkbox`、`select`，等等) 传送窗体到服务器端时，所使用的编码方式是 `application/x-www-form-urlencoded`，但是若要传送文件至服务器端时，必须使用 `multipart/form-data` 的编码方式。正因为双方在传送数据时所使用的编码方式不一样，因此在接收用户传来的数据时，不能直接使用 `request.getParameter()` 来取得。假若读者有兴趣了解文件上传的规范，可以参阅下列这个网址 <http://www.ietf.org/rfc/rfc1867.txt>，将可以查到更加详细的规范说明。

9-4-2 解决文件上传的问题

了解了两者之间的差异后，接下来就说明解决之道。首先，目前有三种解决方案使用较广，一家是 `jspsmart` 公司，另一家是欧莱礼 (O'Reilly)，最后一家是 Jakarta Apache 的 `commons FileUpload`。

首先介绍 `jspsmart` 公司，它推出一套名为 `jspSmartUpload` 的组件，安装好它的组件后，就可以处理文件上传的问题，笔者也曾经试用过这套组件，使用起来简单好用，而且它可供免费下载使用，网址如下：<http://www.jspsmart.com> (注：编辑本书之时，该网站已经关闭)。不过，它并没有公开源程序，这是它惟一可惜的一点 (见图 9-3)。

然后介绍由欧莱礼 (O'Reilly) 公司所提供的 `MultipartRequest` 类工具，它除了可供免费下载之外，还会不定期新增功能，并且它还有公开源代码供用户参考，有兴趣的读者可至：<http://www.servlets.com/cos/index.html> (见图 9-4)。

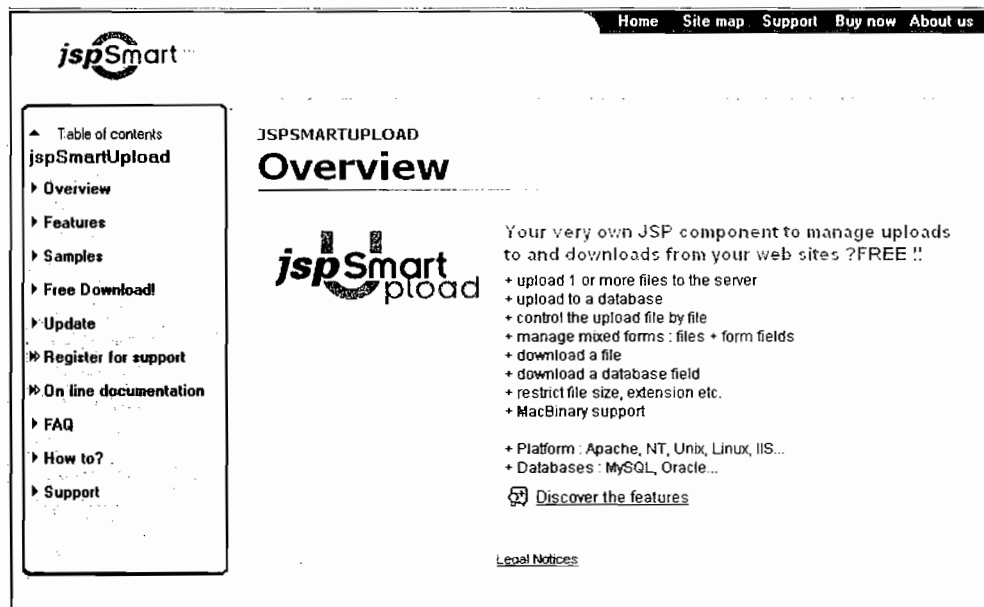


图 9-3 jspSmartUpload 网页

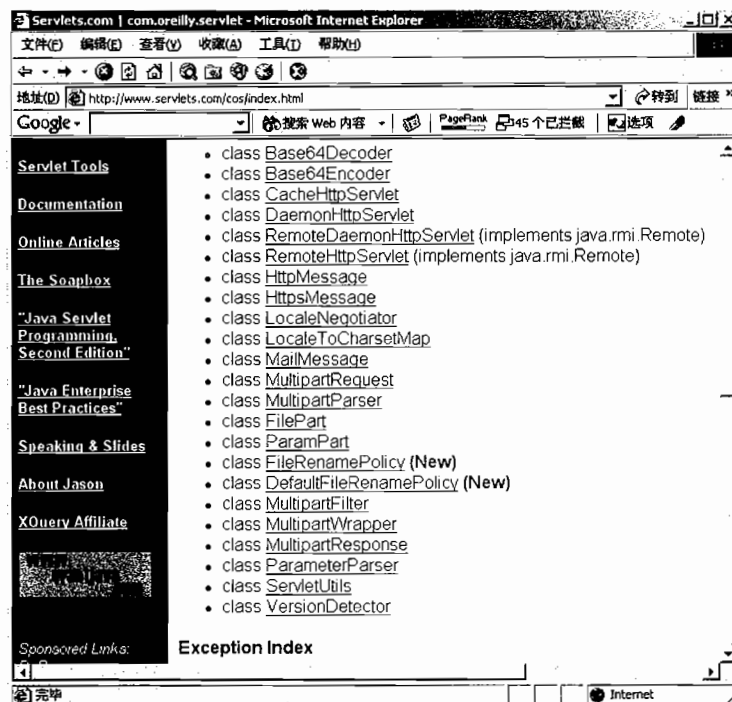


图 9-4 http://www.servlets.com/cos/index.html 网页

9-4-3 欧莱礼上传套件

免费下载或者直接取用本书 CD 内附之版本，笔者收录的版本是 *cos-05Nov2002.zip*。

下面我们将举一个范例来让读者了解如何使用 `MultipartRequest` 类工具，来实现文件上传的功能。

首先我们介绍 `com.oreilly.servlet.MultipartRequest` 这个类工具：

`MultipartRequest` 有八个构造函数(Constructor)：

```
public MultipartRequest(HttpServletRequest request,
                        String saveDirectory) throws IOException

public MultipartRequest(HttpServletRequest request,
                        String saveDirectory,
                        int maxPostSize ) throws IOException

public MultipartRequest(HttpServletRequest request,
                        String saveDirectory, int maxPostSize,
                        FileRenamePolicy policy ) throws IOException

public MultipartRequest(HttpServletRequest request,
                        String saveDirectory, int maxPostSize,
                        String encoding ) throws IOException

public MultipartRequest(HttpServletRequest request,
                        String saveDirectory, int maxPostSize,
                        String encoding, FileRenamePolicy policy )
                        throws IOException

public MultipartRequest(HttpServletRequest request,
                        String saveDirectory,
                        String encoding) throws IOException

public MultipartRequest(ServletRequest request,String saveDirectory)
                        throws IOException

public MultipartRequest(ServletRequest request,
                        String saveDirectory, int maxPostSize)
                        throws IOException
```

上面这八个构造函数都会产生新的 `MultipartRequest` 对象，不过我们通常都是使用前六个构造函数，因为它们皆是用来专门处理 HTTP 协议的请求的。`saveDirectory` 是上传文件所要储存在服务端的目录名称；`maxPostSize` 是限制用户上传文件的大小，若超过 `maxPostSize`，会产生 `IOException`，默认上传文件大小为 1MB；`encoding` 可以设定用何种编码方式来编码上传文件之名称。

`MultipartRequest` 类工具有八种方法，利用这些方法，程序员可以取得请求的相关信息。以下列出八种方法：

JSP2.0 技术手册


```
public Enumeration MultipartRequest.getParameterNames( )
public Enumeration MultipartRequest.getFileNames( )
public String MultipartRequest.getParameter(String name)
public String [ ] MultipartRequest.getParameterValues(String name)
public String MultipartRequest.getFilesystemName(String name)
public String MultipartRequest.getContentType(String name)
public File getFile(String name)
public String getOriginalFileName(String name)
```

接下来就一一为读者介绍这八种方法：

```
public Enumeration getParameterNames( )
```

你可以利用 `getParameterNames()` 的方法来取得所有请求参数的名称，此方法会以 `String` 对象组成的 `Enumeration` 传回所有参数的名称；若没有参数时，传回空的 `Enumeration`。

```
public String getParameter(String name)
```

假若要知道参数值时，可以使用 `getParameter()`，此方法会回传参数为 `name` 的值，类型为 `String`。若没有参数名称 `name` 时，会回传 `null`。若指定参数具有多个值时，只会回传最后一个值，因此假若你要取得所有的值时，就必须使用 `getParameterValues()`。如下所示：

```
public String[ ] getParameterValues(String name)
```

此方法主要用在取得当一指定参数具有多个值时，它会回传 `String` 的数组。若没有这个参数名称 `name` 时，会回传 `null`。

```
public Enumeration getFileNames( )
```

你可以利用 `getFileNames()` 传回所有文件输入类型的名称。此方法会以 `String` 对象组成的 `Enumeration` 传回所有文件输入类型的名称。若没有上传文件时，会回传空的 `Enumeration`。不过请注意，这里所指的文件输入类型的名称，是指在 HTML 表格中，`<input type="file" name="xxx">`，`xxx` 就是文件输入类型的名称，并不是真正上传文件的文件名称，这是要特别小心的地方。

```
public String getFilesystemName(String name)
```

你必须使用 `getFilesystemName()` 才能真正得到上传文件的文件名称，此方法回传 `String` 类型的文件名称。这里的 `name` 是指文件输入类型的名称。

```
public String getContentType(String name)
```

如果想要取得上传文件的内容类型时，你就必须使用 `MultipartRequest.getContentType()` 方法。

```
public File getFile(String name)
```

最后，你可以用 `getFile()` 得到该文件的 `java.io.File` 对象。此方法会传回一个 `File` 的对象，代表储存在服务器上的 `name` 文件。

介绍完 `com.oreilly.servlet.MultipartRequest` 的构造函数和方法之后，笔者就利用这个类工具来做一个范例，希望读者看完这个范例之后，将来在处理文件上传时，能够有所帮助。

这个范例会有两个文件，名称皆为 `File`，不过一个是 `HTML` 文件，另一个是 `JSP` 文件。

■ `File.html`

```
<html>
<head>
  <title>CH9 - File.html</title>
<meta http-equiv="Content-Type" content="text/html; charset=GB2312">
</head>
<body>

<h2>文件上传 - O'Reilly</h2>
<form name="Form1" enctype="multipart/form-data" method="post"
  action="File.jsp">
  <p>上传文件 1:
    <input type="file" name="File1" size="20" maxlength="20">
  </p>
  <p>文件 1 描述:
    <input type="text" name="File1" size="30" maxlength="50">
  </p>
  <p>上传文件 2:
    <input type="file" name="File2" size="20" maxlength="20">
  </p>
  <p>文件 2 描述:
    <input type="text" name="File2" size="30" maxlength="50">
  </p>
  <p>上传文件 3:
    <input type="file" name="File3" size="20" maxlength="20">
  </p>
  <p>文件 3 描述:
    <input type="text" name="File3" size="30" maxlength="50">
  </p>
  <p>
    <input type="submit" value="上传">
    <input type="reset" value="清除">
  </p>
</form>

</body>
</html> </html>
```

执行结果如图 9-5 所示，读者可看出，在 `File.html` 中，我们允许用户一次上传三个文件，并且可以针对每个上传文件做一小段的叙述。

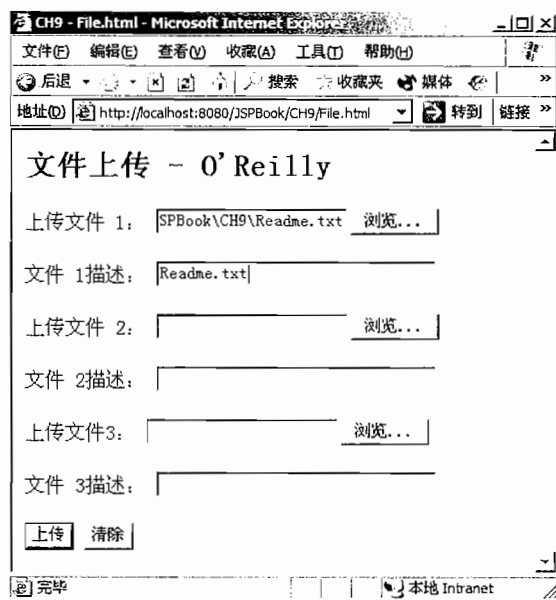


图 9-5 File.html 的执行结果

下面再来看 *File.jsp*。

■ File.jsp

```
<%@ page import="java.io.*" %>
<%@ page import="java.util.*" %>
<%@ page import="com.oreilly.servlet.MultipartRequest" %>
<%@ page contentType="text/html; charset=GB2312" %>

<html>
<head>
    <title>CH9 - File.jsp</title>
</head>
<%
    // 声明将上传的文件放置到服务器的 C:\Upload 目录中
    // 声明限制上传的文件大小为 5 MB
    String saveDirectory = "C:\\Upload\\";
    int maxPostSize = 5 * 1024 * 1024 ;

    // 声明上传文件名称
    String FileName = null;

    // 声明上传文件类型
    String ContentType = null;

    // 声明上传文件内容描述
    String Description = null;
```

JSP2.0 技术手册

```

// 计算上传文件的个数
int count = 0 ;

// 产一个新的 MultipartRequest 的对象, multi
MultipartRequest multi = new MultipartRequest(request, saveDirectory,
                                              maxPostSize );

%>
<body>
<%
// 取得所有上传的文件输入类型名称及描述
Enumeration filename = multi.getFileNames();
Enumeration filesdc = multi.getParameterNames();

while (filename.hasMoreElements())
{
    String name = (String)filename.nextElement();
    String dc = (String)filesdc.nextElement();
    FileName = multi.getFilesystemName(name);
    ContentType = multi.getContentType(name);
    Description = multi.getParameter(dc);

    if (FileName != null)
    {
        count ++;
    }
}

%>
<font color="red">您上传的第<%= count %>个文件: </font><br>
文件名称为: <%= FileName %><br>
文件类型为: <%= ContentType %><br>
文件的描述: <%= Description %><br><br>

<%
    } // end if
} // end while
%>
您总共上传<font color="red"><%= count %></font>个文件
</body>
</html>

```

File.jsp 中需要用到 `java.io.File` 类, 因此我们必须导入 `java.io.*`; 程序中还会用到 `Enumeration` 类, 所以导入 `java.util.*`; 最后导入 `com.oreilly.servlet.MultipartRequest`, 是为了能够处理文件上传的功能。

```

<%@ page import="java.io.*" %>
<%@ page import="java.util.*" %>
<%@ page import="com.oreilly.servlet.MultipartRequest" %>

```

做好这些前置工作之后, 再来就是产生 `MultipartRequest` 对象, 如下所示:

```

MultipartRequest multi = new MultipartRequest (
    request, saveDirectory, maxPostSize );

```

JSP2.0 技术手册

我们也定义好上传文件要放置的目录位置，还有限制上传文件的大小。目前笔者将之定义如下：

```
String saveDirectory = "C:\\\\Upload\\\\";  
int maxPostSize = 5 * 1024 * 1024 ;
```

表示笔者将上传的文件放置到 *C:\\Upload* 目录中，并且限制上传文件大小为 5MB。

注意

你所定义要放置的目录必须事前建立，不然会产生错误信息。

再来利用 `multi.getFileNames()` 取得上传的文件输入类型名称的 `Enumeration` 类型对象，然后使用 `multi.getParameterNames()` 取得上传文件描述的 `Enumeration` 类型对象，最后再将上述两者 `Enumeration` 对象的内容一一取出来，这些信息当中包括：文件名称、文件类型、文件的描述。以下是整个 *File.jsp* 最主要的程序代码：

```
// 取得所有上传的文件输入类型名称及描述  
Enumeration filename = multi.getFileNames();  
Enumeration filesdc = multi.getParameterNames();  
  
while (filename.hasMoreElements())  
{  
    String name = (String)filename.nextElement();  
    String dc = (String)filesdc.nextElement();  
    FileName = multi.getFilesystemName(name);  
    ContentType = multi.getContentType(name);  
    Description = multi.getParameter(dc);  
}
```

开始我们先取得所有文件输入类型的名称，即 `<input type="file" name="xxxx" >` 中的 `xxxx`，因此根据 *File.html* 的内容，`filename` 的内容分别由 *File1*、*File2* 和 *File3* 组成。然后使用 `filename.nextElement()` 将它们一一取出来，存入到 `name` 的变量当中，而 `name` 的内容就是 *File1* 或 *File2* 或 *File3*，最后再利用 `name`，取得真正上传文件名称、文件类型、文件的描述。图 9-6 是 *File.jsp* 的执行结果。

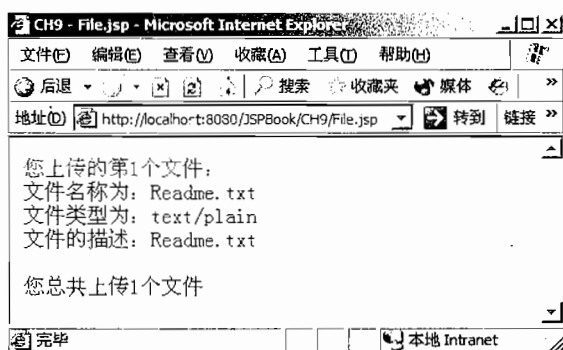


图 9-6 File.jsp 的执行结果

最后我们来看看 C:\Upload 目录下产生的一个 Readme.txt 文件，如图 9-7 所示。

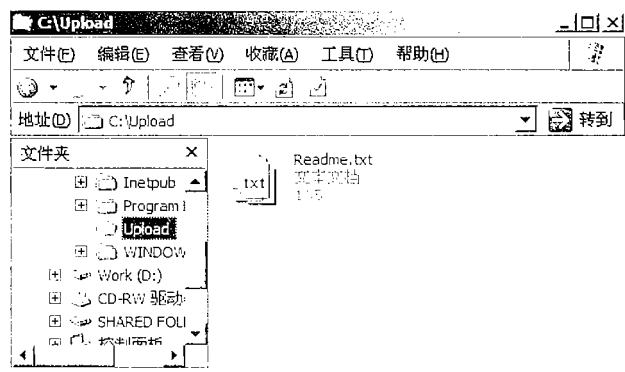


图 9-7 执行 File.jsp 后，C:\Upload 目录下的内容

9-4-4 解决上传中文名文件的问题

前面我们使用 *File.jsp* 来上传非中文名的文件，都不会发生任何的问题，但是假若要上传中文名的文件时会变成乱码，因此接下来笔者将介绍两种方法，可以正确地上传中文名的文件。这两种方法都非常简单：第一种是对 *File.jsp* 做小幅度的修改，让它能够支持上传中文名文件的功能；第二种使用一开始介绍的 jspsmart 公司所提供免费使用的上传文件组件 (jspSmartUpload)。因此接下来我们将做两个简单的范例程序来说明如何解决中文名的问题。

首先我们改写之前的范例程序，一样有两个文件：*File_Chinese.html* 和 *File_Chinese.jsp*。其中 *File_Chinese.html* 和 *File.html* 的内容大部分都一样，只有 form action 改为 *File_Chinese.jsp*，如下所示：

■ *File_Chinese.html*

```
..... 略
<form name="Form1" enctype="multipart/form-data" method="post"
action="File_Chinese.jsp">
..... 略
```

然后就是 *File_Chinese.jsp*，它只比 *File.jsp* 多一个设定编码，片段程序如下：

■ *File_Chinese.jsp*

```
<%!
// 声明将上传的文件放置到服务器的 C:\Upload 目录中
// 声明限制上传的文件大小为 5 MB

String saveDirectory = "C:\\\\Upload\\";
int maxPostSize = 5 * 1024 * 1024 ;

// 声明上传文件名所使用的编码，默认值为 ISO-8859-1，
// 若改为 GB2312 则支持中文名
```



```
String enCoding = " GB2312";
..... 略
%>
..... 略
<%
// 产生一个新的 MultipartRequest 对象, multi
MultipartRequest multi = new MultipartRequest(request, saveDirectory,
maxPostSize, enCoding);
%>
..... 略
```

即在 new 一个新的 MultipartRequest 对象时, 多设定一个参数 enCoding。因为我们要支持简体中文, 因此可设为 GB2312, 同样地, 假若要支持繁体中文的话, 只要改为 MS950 或 Big5 即可。图 9-8 为执行 File_Chinese.html 上传一个“博文视点.doc”的文件, 而图 9-9 为执行 File_Chinese.jsp 后, C:\Upload 目录的结果。

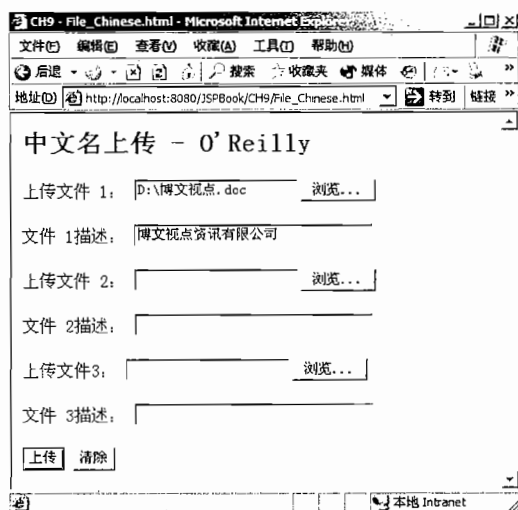


图 9-8 上传一个“博文视点.doc”的文件

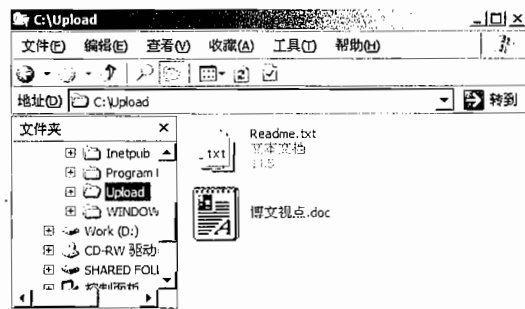


图 9-9 执行 File_Chinese.jsp 后, C:\Upload 目录的结果

9-5 jspSmartUpload——上传和下载

接下来介绍使用 jspsmart 公司所免费提供的 jspSmartUpload 上传组件，读者可自行至 <http://www.jspsmart.com/> 下载，或者直接取用光盘上的文件，名称为：*jspSmartUpload.zip*。

首先我们来介绍 com.jspsmart.upload.SmartUpload、com.jspsmart.upload.Files、com.jspsmart.upload.File、com.jspsmart.upload.File 这几个类工具：

SmartUpload 只有一个构造函数，如下所示：

```
public SmartUpload( )
```

jspSmartUpload 内附的范例中它都是使用 JavaBean 的方式来产生一个新的 SmartUpload 对象，如下：

```
<jsp:useBean id="mySmartUpload" scope="page"
  class="com.jspsmart.upload.SmartUpload" />
```

SmartUpload 有许多的方法，通过这些方法，程序员可以取得上传处理的相关设定。以下列出比较常用的方法：

```
public final void initialize(javax.servlet.jsp.PageContext pageContext)
public void downloadFile(String sourceFilePathName)
public int getSize( )
public Files getFiles( )
public Request getRequest( )
public void setDeniedFilesList(String deniedFilesList)
public void setAllowedFilesList(String allowedFilesList)
public void setTotalMaxFileSize(long totalMaxFileSize)
public void setMaxFileSize(long maxFileSize)
public int save(String destPathName)
public void upload( )
```

接下来就为读者介绍这几个方法：

```
public final void initialize(javax.servlet.jsp.PageContext pageContext)
```

首先当声明完 SmartUpload 对象后，我们必须使用 initialize()，并放入 pageContext 来完成初始化的动作。

```
public void downloadFile(String sourceFilePathName)
```

下载文件时可以使用 downloadFile()，填入文件存在的位置后就可以下载文件。

```
public int getSize( )
```

如果想要知道上传了多少大小的文件时，使用 getSize() 就可以得到所有上传文件大小的总和，此方法将回传 int 类型的文件大小。

```
public Files getFiles( )
```

使用这个方法将会回传上传文件的集合，回传值的类型是 `com.jspsmart.upload.Files`。

```
public Request getRequest( )
```

使用 `getRequest` 这个方法将会回传 `Request`，要注意的是这里的 `Request` 的类型是 `com.jspsmart.upload.Request`。

```
public void setDeniedFilesList(String deniedFilesList)
```

你可以利用 `setDeniedFilesList()` 来限定用户上传的文件类型，这个方法是设定不可以上传的文件类型。例如：`setDeniedFilesList("txt")` 上传扩展名为 `txt` 的文件时将会产生 `java.lang.SecurityException` 的错误。另外为了防止上传没有扩展名的文件，可以新增两个`","`，如：`","`，让 `SmartUpload` 没办法上传没有扩展名的文件。

```
public void setAllowedFilesList(String allowedFilesList)
```

`setAllowedFilesList()` 用来设定可以上传的文件类型例如：`setAllowedFilesList("htm,html,txt")`，那么将只能上传 `html` 或纯文本文件。另外为了可以上传没有扩展名的文件，可以新增两个`","`，如：`","`，让 `SmartUpload` 允许上传没有扩展名的文件。

```
public void setTotalMaxFileSize(long totalMaxFileSize)
```

你可以使用 `setTotalMaxFileSize()` 来限制上传文件的大小，如同它字面上的意思，这是限制每一次上传时所有文件的大小总和。如果传入的参数是 `0`，那么将不会限制上传的大小。

```
public void setMaxFileSize(long MaxFileSize)
```

`setMaxFileSize()` 用来限制用户上传文件的大小，跟上面相反。这个方法是限制每个文件的最大值。如果传入的参数是 `0`，那么将不会限制上传的文件大小。

```
public int save(String destPathName)
```

当上传完成后，要调用 `save()` 才能将文件存放于指定位置中。这里的 `destPathName` 是指 `server` 端存放文件的路径名称。例如：要存放于 `C:\Upload`，那么就要像这样调用 `save("C:\\\\Upload\\")`，此方法将回传上传成功的文件个数，类型为 `int`。

```
public void upload( )
```

当初始化也就是 `initialize()` 完成后，必须调用 `upload()` 来执行上传动作。这个动作主要是将所有 `Form` 中填的数据上传。

接下来我们看看 `com.jspsmart.upload.Files`。在文件中它提到我们不能直接去声明 `Files` 对象，必须从 `SmartUpload` 的 `getFiles()` 得到。以下列出 `Files` 中比较常用的几个方法：

```
public int getSize( )
public Files getFile( )
public int getCount( )
```

接下来看看这几个方法：

```
public long getSize( )
```

如果想要知道到底上传了多大的文件时，`getSize()` 将会回传 `Files` 中所有文件大小的总和。

```
public File getFile(int index)
```

依据 index 的值回传对应的 File 对象。注意这里回传的 File 为 com.jspmart.upload.File。

```
public int getCount( )
```

你可以使用这个方法来得得到 Files 中的文件个数。

接下来是 com.jspmart.upload.File。在文件中它提到我们不能直接去声明 File 对象，它必须从 Files 的 getFile() 得到。以下列出 File 中比较常用的几个方法：

```
public void saveAs(String destFilePathName)
public boolean isMissing( )
public String getFieldName( )
public String getFilePathName( )
public String getFileExt( )
public String getContentType( )
public int getSize( )
```

接下来一一介绍这几个方法：

```
public void saveAs(String destFilePathName)
```

这个方法是将文件存放于 destFilePathName 所指定的位置。

```
public boolean isMissing( )
```

这个方法是测试文件是否确实已存在，如果文件确实没有上传，那么将会回传 true。

```
public String getFieldName( )
```

我们可以使用这个方法来得得到此 File 在前一个 html 文件中的表格名称，此方法将回传 String 类型的表格名称。

```
public String getFilePathName( )
```

你可以使用这个方法来得得到此 File 在上传端的文件位置，此方法将回传 String 类型的文件路径位置。

```
public String getFileExt( )
```

当我们想得到文件的类型时，可以使用这个方法来得得到上传文件的扩展名，此方法将回传 String 类型的扩展名。

```
public String getContentType( )
```

如果你想要取得上传文件的内容类型时，你就必须使用 getContentType()。

```
public int getSize( )
```

当我们想得到各个文件大小时，可以使用 getSize() 来得得到每个上传文件的大小，此方法将回传类型为 int 的文件大小。

最后我们来介绍一下 com.jspmart.upload.Request。在文件中它提到我们不能直接去声

明 Request 对象，它必须从 SmartUpload 的 getRequest() 得到。以下列出 Request 中的三个方法：

```
public String getParameter(String name)
public Enumeration getParameterNames()
public String [] getParameterValues(String name)
```

下面介绍这几个方法：

```
public String getParameter(String name)
```

假若想知道参数值，可以使用 getParameter()，此方法会回传参数为 name 的值，类型为 String。若没有参数名称 name 时，会回传 null。若指定参数具有多个值时，只会回传最后一个值，因此假若你要取得所有的值时，你必须使用 getParameterValues()。

```
public Enumeration getParameterNames()
```

你可以利用 getParameterNames() 的方法来取得所有请求参数的名称，此方法会以 String 对象组成的 Enumeration 传回所有参数的名称；若没有参数时，传回空的 Enumeration。

```
public String [] getParameterValues(String name)
```

此方法主要用在取得当一指定参数具有多个值时，它会回传 String 的数组。若没有这个参数名称 name 时，会回传 null。

接下来笔者将写几个范例，让大家慢慢了解如何使用 jspSmartUpload。

第一个范例是最简单的做法，整个 JSP 只用到 SmartUpload 这个类，所以会发现根本没有几行程序代码。首先我们在 Jspsmart1.html 选择用户要上传的文件，然后再交由 Jspsmart1.jsp 去处理。

■ Jspsmart1.html

```
<html>
<head>
  <title>CH9 - Jspsmart1.html</title>
  <meta http-equiv="Content-Type" content="text/html; charset=GB2312">
</head>
<body>

<h2>文件上传范例 - jspSmart</h2>

<form name="Form1" enctype="multipart/form-data" method="post"
  action="Jspsmart1.jsp">
<p>上传文件 1: <input type="file" name="File1" size="20" maxlength="20"></p>
<input type="submit" value="上传">
<input type="reset" value="清除">
</form>

</body>
</html>
```

■ Jspsmart1.jsp

```
<%@ page import="com.jspsmart.upload.*" %>
<%@ page contentType="text/html; charset=GB2312" %>
```



```

<html>
<head>
  <title>CH9 - Jspsmart1.jsp</title>
</head>
<body>

<h2>文件上传范例 - jspSmart</h2>

<jsp:useBean id="mySmartUpload" scope="page"
  class="com.jspsmart.upload.SmartUpload" />
<%
    //计算文件上传个数
    int count=0;

    //SmartUpload的初始化,使用这个 jspsmart 一定要在一开始就这样声明
    mySmartUpload.initialize(pageContext);

    //声明限制上传的文件大小为 5 MB
    mySmartUpload.setMaxFileSize(5 * 1024 * 1024);

    //依据 form 的内容上传
    mySmartUpload.upload();

    try {
        //将文件存放于 C:\Upload\
        count = mySmartUpload.save("C:\\Upload\\");

        //显示上传文件个数
        out.println("您成功上传"+count + "个文件.");
    } catch (Exception e) {
        out.println(e.toString());
    }
%>
</body>
</html>

```

在 *Jspsmart1.jsp* 中我们主要用到的是 `com.jspsmart.upload.SmartUpload` 类, 所以在一开始我们必须导入它。

```
<%@ page import="com.jspsmart.upload.*" %>
```

做好这些前置工作之后, 然后就是产生 `SmartUpload` 的对象, 如下所示:

```
<jsp:useBean id="mySmartUpload" scope="page"
  class="com.jspsmart.upload.SmartUpload" />
```

接下来笔者将限制上传文件的大小, 并做上传的动作。如下:

```
mySmartUpload.setMaxFileSize(5 * 1024 * 1024);
mySmartUpload.upload();
```

如上所示, 笔者将上传文件的总大小设定为 5MB。要注意的是, 它和 oreilly 的

MultipartRequest 不太一样，必须在 upload 后才能将文件存放于指定位置。

最后将文件存放于 *C:\Upload* 中，并且得到文件数目的值。

```
count = mySmartUpload.save("C:\\Upload\\");
```

接下来再看下一个范例，我们将上传文件并且得到各个文件的详细内容，并在 *Jspsmart2.jsp* 中显示出来。因为 *Jspsmart2.html* 和之前的大同小异，所以这里我们跳过 *Jspsmart2.html* 直接看 *Jspsmart2.jsp*。

■ Jspsmart2.jsp

```
<%@ page import="com.jspsmart.upload.*" %>
<%@ page contentType="text/html; charset=GB2312" %>

<html>
<head>
<title>CH9 - Jspsmart2.jsp</title>
</head>
<body>

<h2>文件上传范例 - jspSmart</h2>

<jsp:useBean id="mySmartUpload" scope="page"
class="com.jspsmart.upload.SmartUpload" />
<%
    //计算文件上传个数
    int count=0;

    //SmartUpload 的初始化,使用这个 jspsmart 一定要在一开始就这样声明
    mySmartUpload.initialize(pageContext);

    //依据 form 的内容上传
    mySmartUpload.upload();

    //将上传的文件一个一个取出来处理
    for (int i=0;i<mySmartUpload.GetFiles().getCount();i++)
    {
        //取出一个文件
        com.jspsmart.upload.File myFile =
            mySmartUpload.GetFiles().getFile(i);

        //如果文件存在,则做存档操作
        if (!myFile.isMissing()) {

            //将文件存放于绝对路径的位置
            myFile.saveAs("C:\\upload\\" + myFile.getFileName(),
                mySmartUpload.SAVE_PHYSICAL);

            //显示此上传文件的详细信息
            out.println("FieldName = " + myFile.getFieldName() + "<BR>");
            out.println("Size = " + myFile.getSize() + "<BR>");
        }
    }
}
```

JSP2.0 技术手册

```

        out.println("FileName = " + myFile.getFileName() + "<BR>");
        out.println("FileExt = " + myFile.getFileExt() + "<BR>");
        out.println("FilePathName = " + myFile.getFilePathName() +
            "<BR>");
        out.println("ContentType = " + myFile.getContentType() +
            "<BR>");
        out.println("ContentDisp = " + myFile.getContentDisp()
            + "<BR>");
        out.println("TypeMIME = " + myFile.getTypeMIME() + "<BR>");
        out.println("SubTypeMIME = " + myFile.getSubTypeMIME() +
            "<BR>");
        count ++;
    }
}

// 显示应该上传的文件数目
out.println("<BR>" + mySmartUpload.getFiles().getCount() +
    " files could be uploaded.<BR>");

// 显示成功上传的文件数目
out.println(count + "file(s) uploaded.");
%>
</body>
</html>

```

Jspsmart2.jsp 中我们使用到 `com.jspsmart.upload.Files` 和 `com.jspsmart.upload.File`。

首先我们利用 `SmartUpload.getFiles()` 得到 `com.jspsmart.upload.Files` 对象, 这个 `Files` 中记录着所有上传的文件。然后再使用 `mySmartUpload.getFiles().getCount()` 得到 `Files` 中到底记录多少个文件。最后我们使用 `mySmartUpload.getFiles().getFile()` 将每个文件的信息一个一个取出来, 存入 `myFile` 中。如下:

```

for (int i=0;i<mySmartUpload.getFiles().getCount();i++){
    com.jspsmart.upload.File myFile =
        mySmartUpload.getFiles().getFile(i);
    //省略
}

```

接下来也是这个范例最主要的地方, 先检查取出的 `File` 是否确实存在, 如果存在, 则将文件存放于 `upload` 这个文件夹中, 并且显示出这个文件的详细信息。最后再把 `count` 的数目加一。

```

if (!myFile.isMissing())
{
    //将文件存放于绝对路径的位置
    myFile.saveAs("C:\\\\Upload\\" + myFile.getFileName(),
        mySmartUpload.PHYSICAL);

    //显示此上传文件的详细信息
    out.println("FieldName = " + myFile.getFieldName() + "<BR>");
    out.println("Size = " + myFile.getSize() + "<BR>");
}

```

JSP2.0 技术手册

```

        out.println("FileName = " + myFile.getFileName() + "<br>");
        out.println("FileExt = " + myFile.getFileExt() + "<br>");
        out.println("FilePathName = " + myFile.getFilePathName() + "<br>");
        out.println("ContentType = " + myFile.getContentType() + "<br>");
        out.println("ContentDisp = " + myFile.getContentDisp() + "<br>");
        out.println("TypeMIME = " + myFile.getTypeMIE() + "<br>");
        out.println("SubTypeMIME = " + myFile.getSubTypeMIME() +
            "<br>");
        count ++;
    }
}

```

读者应该发现存档操作和先前有点不一样了。这次我们使用的是 `myFile.saveAs()`，将文件存放于 `C:\Upload` 下。

```

myFile.saveAs("C:\\ Upload\\" + myFile.getFileName());
myFile.saveAs("C:\\ Upload\\" + myFile.getFileName(),
    mySmartUpload.SAVE_PHYSICAL);

```

上面这两种方法其实代表的意义相同，如果读者想自行改变上传后的文件名，可以将上面这两行的 `myFile.getFileName()` 改成自己想要的名称即可。

接下来笔者从【桌面】上传一个 eclipse 的 word 文件至 `C:\Upload`，则 `Jspsmart2.jsp` 的执行画面如图 9-10 所示：

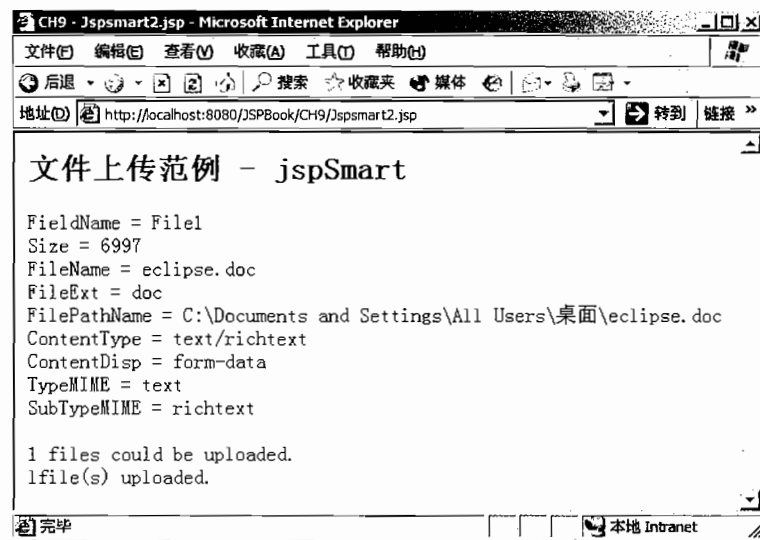


图 9-10 Jspsmart2.jsp 的执行结果

看完上面几个范例，应该已了解一些 `JspSmartUpload` 的基本应用了。接下来看其他内附的功能，下面这个 `Jspsmart3.jsp` 就是用来限制用户上传文件的类型和限制存取的位置。使用的 html 仍然和之前相同。

■ Jspsmart3.jsp

```
<%@ page import="com.jspsmart.upload.*" %>
<%@ page contentType="text/html; charset=GB2312" %>

<html>
<head>
  <title>CH9 - Jspsmart3.jsp</title>
</head>
<body>

<h2>文件上传范例 - jspSmart</h2>

<jsp:useBean id="mySmartUpload" scope="page"
class="com.jspsmart.upload.SmartUpload" />
<%

    //计算文件上传个数
    int count=0;

    //SmartUpload 的初始化,使用这个 jspsmart 一定要在一开始就这样声明
    mySmartUpload.initialize(pageContext);

    //声明可以上传的文件类型
    mySmartUpload.setAllowedFilesList("htm,html,txt,,");

    //限制磁盘档置,可存档于绝对位置
    mySmartUpload.setDenyPhysicalPath(false);

    //依据 form 的内容上传
    mySmartUpload.upload();

    //将文件用原本的名字存放于 server 上的相对路径
    try {
        count = mySmartUpload.save("C:\\upload\\",
            mySmartUpload.SAVE_PHYSICAL);
    } catch (Exception e) {

        out.println("<b>Wrong selection : </b>" + e.toString());
    }

    //显示总共上传文件个数
    out.println(count + " file(s) uploaded.");
%>

</body>
</html>
```

设定和之前都差不多,我们直接看新增的部分。这个程序主要新增了两个地方。使用 mySmartUpload.setAllowedFilesList() 来允许哪些文件是可以上传的,在 Jspsmart3.jsp 中

允许可以上传的文件有 *htm*、*html*、*txt* 及没有扩展名的文件。接下来使用 `mySmartUpload.setDenyPhysicalPath()` 来限制用户是否可以将文件存于绝对路径。

```
mySmartUpload.setAllowedFilesList("htm,html,txt,,");
mySmartUpload.setDenyPhysicalPath(false);
```

如果不使用 `mySmartUpload.setDenyPhysicalPath()`，则默认就是可以指定存到绝对路径下。笔者在上面范例中写入这一行是希望读者可以自己试试看，把参数改成 `true` 以后，将会发现无法指定存档于绝对路径下。

接下来这部分虽然不属于上传部分，但是因为 `jspSmartUpload` 提供下载功能，所以笔者在这里简单介绍一下使用 `jspSmartUpload` 的下载方法。

在 `jspSmartUpload` 中，提供以下几个方法来 `download`。

```
public void downloadFile( String sourceFilePathName)
public void downloadFile( String sourceFilePathName,
                          String contentType)
public void downloadFile( String sourceFilePathName,
                          String contentType,
                          String destFileName)
public void downloadFile( String sourceFilePathName,
                          String contentType,
                          String destFileName,
                          int    blockSize)
public void downloadField( ResultSet rs,
                          String columnName,
                          String contentType,
                          String destFileName)
```

接下来的范例中，我们使用下面这个方法，`sourceFilePathName` 代表服务端的文件地址，`contentType` 代表的是文件类型，而 `destFileName` 代表的是下载时的默认储存名称。

```
public void downloadFile(String sourceFilePathName,
                          String contentType,
                          String destFileName)
```

我们来实现一个范例 `download.jsp`，这个范例主要是下载 `C:\Upload\` 下的 `sample.zip`。

■ `download.jsp`

```
<%@ page import="com.jspsmart.upload.*" %>
<%@ page contentType="text/html; charset=GB2312" %>

<html>
<head>
  <title>CH9 - download.jsp</title>
</head>
<body>
```



```
<h2>文件下载范例 - jspSmart</h2>

<jsp:useBean id="mySmartUpload" scope="page"
    class="com.jspsmart.upload.SmartUpload" />

<%
    // SmartUpload 的初始化
    mySmartUpload.initialize(pageContext);

    //必须如此声明, 否则将会把文件显示于浏览器中
    mySmartUpload.setContentDisposition("inline;");

    //将 sample.zip 下载, 下载默认名称为 downloaded.zip
    mySmartUpload.downloadFile("C:\\upload\\sample.zip",
        "application/x-zip-compressed",
        "downloaded.zip");
%>

</body>
</html>
```

接着只要指定到这个 jsp, 就会跳出下载窗口, 并问用户是否下载文件。读者应该可以发现, 这里加了一行:

```
mySmartUpload.setContentDisposition("inline;");
```

这是为了让下载窗口跳出, 否则将会发现浏览器开始显示乱码, 无法下载并保存文件。

上面这么多范例其实都是从 JspSmartUpload 中内附的范例修改而来的。在此笔者强烈建议读者能好好观看上面的几个范例, 熟悉一下它的操作方法, 相信就能很快地上手, 并且最重要的一点, 它还能支持中文名, 十分的便利。

9-6 本文区输入类型 (Textarea)

本文区输入类型会产生一个由数行本文输入类型(text)所构成的区域, 通常用于须要输入大量的文字。

```
<TEXTAREA NAME="textarea" COLS="n" ROWS="m">
</TEXTAREA>
```

本文区的属性如表 9-7 所示:

表 9-7

属性名称	描述
NAME	本文区的名称
COLS	本文区的长
ROWS	本文区的宽

笔者将本文区输入类型独立作为一节来介绍, 主要是因为用户在本文区输入的数据内容有

换行时，数据传送至服务器端，再将数据显示出来，原本用户有断行的地方皆不见了，因此本节主要解决这个问题。

用户在本文区输入数据时，换行是表示 ASCII 0x0D。但是当服务端接收到数据时，是以 HTML 格式输出的，因此我们要将 ASCII 0x0D 转为 HTML 格式的换行标签
。接下来就写个范例程序 *Textarea.jsp* 来解决这个问题。

■ Textarea.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>
<html>
<head>
  <title>CH9 - Textarea.jsp</title>
</head>
<body>

<h2>Textarea 范例</h2>
<fmt:requestEncoding value="GB2312" />

<%
  // 将来自 Textarea 的信息，存入 msg 的字符串当中
  String msg = request.getParameter("Message");

  // 声明一新字符串，表示修改过字符串后的结果
  String Message = "";

  // msgLength 表示 Textarea 的字符串长度
  int msgLength = msg.length();

  // Position 表示目前的指针位置
  int Position = 0;

  while (true)
  {
    // 表示 0x0D 的位置
    int index = msg.indexOf(0x0D, Position);

    // 假设都没有换行时，直接离开 while 循环
    if (index == -1) { break; }

    // 假设有换行时，将换行之前的字符串
    // 放置到新的 Message 字符串上，做完再加上 <br>
    if (index > Position) {
      Message += msg.substring(Position, index);
    }

    Message += "<br>";
    Position = index + 1;
  }

  if (Position >= 0) {
    Message += msg.substring(Position);
  }
%>
```

JSP2.0 技术手册

```
        out.println(Message);  
    %>  
  
</body>  
</html>
```

Textarea.jsp 主要就是将 0x0D 转换为
, 因此程序并不难, 只是一些有关 Java 字符串的处理。假若用户不太了解如何处理字符串, 请自行参阅一般 Java 的书籍, 皆有对字符串的详细解说。图 9-11 为 *Textarea.html* 的执行结果, 图 9-12 为 *Textarea.jsp* 的执行结果。

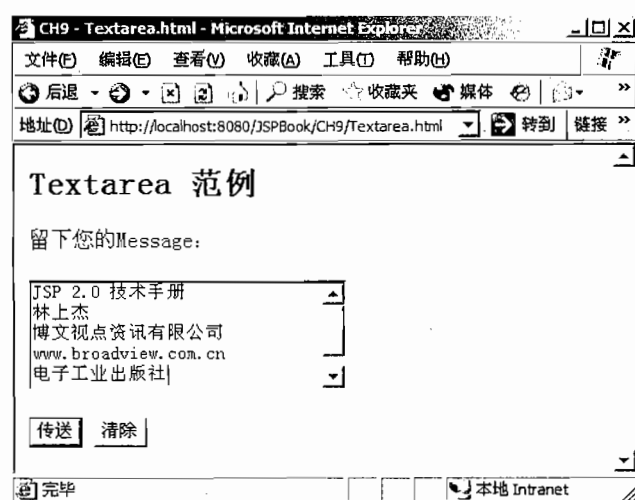


图 9-11 Textarea.html 的执行结果



图 9-12 Textarea.jsp 的执行结果

JSP2.0 技术手册

另外补充一点, JDK 1.4 之后多了一个好用的方法: `replaceAll()`。假若读者安装的是 JDK 1.4 版本, 可以利用 `replaceAll()` 来满足我们的需求, 请看下列的范例: *ReplaceAll.html* 和 *ReplaceAll.jsp*。其中 *ReplaceAll.html* 和 *Textarea.html* 相似, 所以不多加说明, 直接看 *ReplaceAll.jsp*:

■ *ReplaceAll.jsp*

```
<%@ page contentType="text/html; charset=GB2312" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<html>
<head>
  <title>CH9 - ReplaceAll.jsp</title>
</head>
<body>

<h2>Textarea 范例 - 使用 replaceAll()</h2>
<fmt:requestEncoding value="GB2312" />

<%
  String msg = request.getParameter("Message");
  String str = msg.replaceAll("\r\n", "<br>");
  out.println(str);
%>

</body>
</html>
```

各位读者可以发现, 使用 `replaceAll()` 方法之后, 程序代码变得非常简单。`replaceAll()` 是利用正则表达式(Regular Expression)才能做到如此强大的功能, 而正则表达式也是 JDK 1.4 很重要的新增功能, 若读者对此有兴趣, 可以自行阅读 Java for JDK 1.4 的书籍。

10

第十章

Session Tracking

本章将说明如何针对横跨数个 HTTP 请求的用户，提供他们需要的服务。若要顺利完成工作，必须依靠 Session Tracking，因此本章将会详细为读者解说 Session Tracking 的来龙去脉，并且希望读者看完本章后，能够善加运用 Servlet 中所提供的有关 Session Tracking 的 API。本章将分 6 节来介绍：

- 10-1 Stateful & Stateless
- 10-2 Session Tracking 的四种方法
- 10-3 Session 的生命周期
- 10-4 HttpSessionBindingListener 接口
- 10-5 Shopping Cart 范例程序一
- 10-6 Shopping Cart 范例程序二

JSP2.0 技术手册

10-1 Stateful & Stateless

Internet 通讯协议可以分为两大类:Stateful 与 Stateless,两者最大的差别在于 Client 与 Server 之间维持联机上的不同。

举例来说:以 Telnet 与 FTP 而言,它们皆同属于 Stateful 的协议,因为当 Client 连接到 Server 端后,经由相同且持续性的联机来传达各种的操作,等待操作完毕之后,Server 端才切断联机。具有 Stateful 协议的 Server 知道这些请求是来自于相同的 Client 端,换句话说,所谓的 State 是指连接状态,而 Stateful 的连接状态是持续性的联机,并且 Server 将会记得每一个 Client。

另一方面,HTTP 是一种 Stateless 的协议。它只关心请求与响应的状态,当 Client 发出请求时,Server 才会建立连接,一旦 Client 的请求结束,Server 便会中断与 Client 的连接,不会一直与 Client 保持联机的状态。对于简单的 Web 来说,是一个很不错的协议。

如果一个 Client 只是单纯地请求一个文件(HTML 或 GIF),Server 可以响应给 Client,并不须要知道一连串的请求是来自于相同的 Client 或是不同的 Client,而且 Server 也不须要担心 Client 现在是处在连接状态还是已经断线了。但是真实世界并不是如此美好,这样的通讯协议,使得 Server 难以判断所连接的 Client 是否是同一个人。

就举常见的购物车来说:用户将商品加入购物车内,此时用户的请求已经结束,Server 中断与用户的连接,假若当用户要进行结账的时候,Server 却不认得这次的用户是否就是刚才将商品放入购物车的人,因而无法进行结账的动作。当我们在撰写 web 程序的时候,上述问题正是我们所关注的,我们必须想办法将相关的请求结合在一起,并且努力维持用户的状态在 Server 上。

10-2 Session Tracking 的四种方法

JSP 技术中,让 Web 服务器能够追踪用户的状态就叫做 Session Tracking。Session Tracking 的动作,简单地说:就是从上一个请求所传送的数据能维持状态到下一个请求,并且辨认出是相同的 Client 端所发送出来的。Session Tracking 可以经由下列四种方式来完成:

- (1) 建立含有数据的隐藏表格字段(Hidden Form Field);
- (2) 重写包含额外参数的 URL(URL Rewriting);
- (3) 使用持续 Cookies (Persistent Cookies);
- (4) 使用 Servlet 的 HttpSession API。

前三种 Session Tracking 是传统的做法,每种做法皆有其缺点。最后一种方法是目前最常用,也是最有效的解决方案,因此笔者将把讨论重心放在第四种 Session Tracking 方法上。

10-2-1 Session 的定义

在介绍 Session Tracking 的方法之前,我们要先对 Session 的定义有正确的了解。Session,中文译为会话:即在一段时间之内,单一 Client 与 Server 之间一连串相关的交互作用。在会话

JSP2.0 技术手册

之内，可以是一连串的交易，如：以浏览器为基础的 E-mail 服务中，用来确认 E-mail 账号等许多请求。

一个会话之内也有可能包含多个请求指向同一个 JSP 文件，或者是请求各种不同的资源，不过，HTTP 是 Stateless 的通讯协议，Server 将不会知道哪个请求是属于哪个会话，因此以下有两种方法让我们解决此种情形：

(1) 可以让 Client 每次发出请求时能被辨认，然后我们可以储存数据及找回用户在 Server 端储存的相关数据。

(2) 可以传送识别的数据给 Client，让 Client 每次发出的请求中夹带之前识别的数据，传送回来。

目前最常使用第二种方法，如图 10-1 所示，服务器将识别用的数据——session id 连同 cookie 一并传送至 Client：

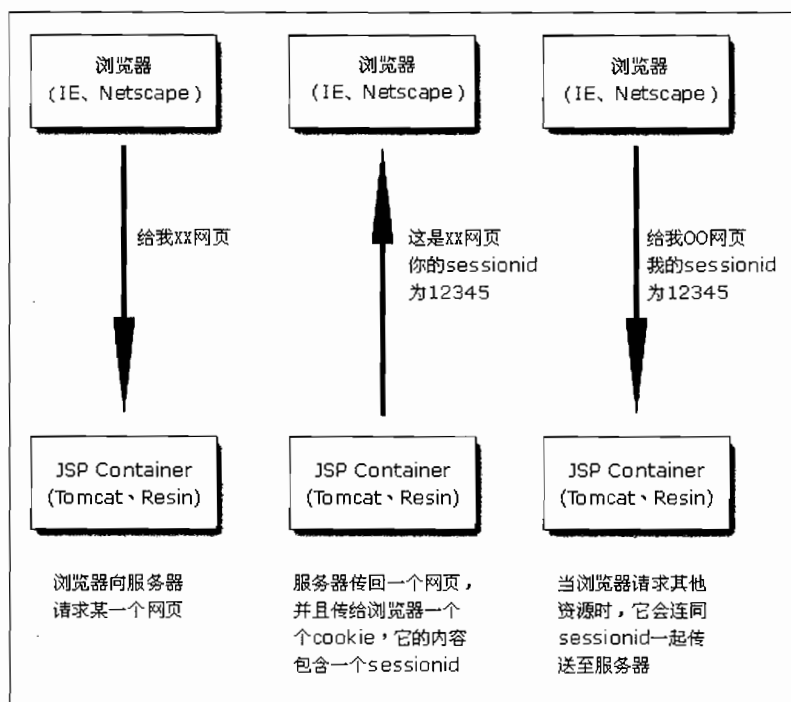


图 10-1 利用 session id 来识别 Client

传统上 HTTP Client 端有几个 Session Tracking 的技巧，如：隐藏字段，URL 重写、Cookie 的机制。目前 JSP 提供 Session Tracking 的方法和类，大幅度地降低了程序员为了追踪用户状态所造成的困扰，然而为求彻底了解 Session Tracking 的机制，笔者还是先将传统式的 Session Tracking 先做一番介绍。

10-2-2 隐藏表格字段(Hidden Form Field)

隐藏表格字段(Hidden Form Field)的方法, 是利用 HTML 内 Hidden 的属性, 把 Client 的信息, 让用户不察觉的情形下, 偷偷地随着请求一起传送到 Server 处理, 这样一来, 就可以进行 Session Tracking 的任务了。你可以下列的方法来做隐藏字段的 Session Tracking:

```
< INPUT TYPE="Hidden" NAME="UserID" value="412313" >
```

然后将重要的用户信息, 如: ID 之类独一无二的信息, 以隐藏字段的方式传送给 Server, Server 取得此字段的值后, 再将用户所传送的内容加入购物车内。隐藏字段的优点在于 Session 数据传送到 Server 端时, 并不用像 GET 的方法, 会将 Session 数据暴露在 URL 之上。不过这种做法还是有它的缺点: 一旦 Session 数据储存在隐藏字段中, 就仍然有暴露数据的危机, 因为只要用户直接观看 HTML 的源文件, Session 数据将会暴露无疑。这将造成安全上的漏洞, 特别当用户数据是依赖用户 ID、密码来取得的时候, 将会有被盗用的危险。

笔者在这里举一个范例, *Hidden.html* 中主要包括两笔隐藏字段, 它们分别为 name 和 number, 其值为 browser 和 1234。当用户按下按钮后, 会将窗体传送到 *Hidden.jsp* 做处理, *Hidden.html* 的完整程序代码如下:

■ Hidden.html

```
<html>
<head>
  <title>CH10 - Hidden.html</title>
  <meta http-equiv="Content-Type" content="text/html; charset=GB2312">
</head>
<body>

  <h2>利用隐藏表格字段的方式传送两笔数据</h2>
  <p>分别传送: </p>
  <p>name = browser</p>
  <p>number = 1234</p>
  <form name="Form" method="post" action="Hidden.jsp">
    <p><input type="hidden" name="name" value="browser" ></p>
    <p><input type="hidden" name="number" value="1234" ></p>

    <input type="submit" value="传送">
  </form>

</body>
</html>
```

当用户按下【传送】按钮时, 就会将数据传送到 *Hidden.jsp* 做处理。*Hidden.jsp* 所做的事情, 只是接收 name 和 number 两个参数, 并且将它们显示出来, 表示成功地利用了隐藏字段来传递参数, 并且也能够顺利取得参数。*Hidden.jsp* 的完整程序如下:

■ Hidden.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>
```

JSP2.0 技术手册

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<html>
<head>
  <title>CH10 - Hidden.jsp</title>
</head>
<body>

<h2>从 Hidden.html 接收到隐藏字段的数据</h2>

name: <font color="red">${param.name}</font><br>
number: <font color="red">${param.number}</font>

<br><a href="/JSPBook/CH7/HelloJSTL.jsp" >Click here</a>

<c:url value="/CH7/HelloJSTL.jsp" var="myUrl">
  <c:param name="name" value="111" />
</c:url>

<br><a href='<c:out value="${myUrl}" />'>Click here</a>

<br><a href='<%= response.encodeURL("/JSPBook/CH7/HelloJSTL.jsp")
%>'>Click here</a>

</body>
</html>
```

Hidden.jsp 的执行结果如图 10-2 所示。



图 10-2 Hidden.jsp 的执行结果

10-2-3 URL 重写(URL Rewriting)

隐藏字段是利用 POST 的方法, 将 Session 数据传给 Server; 而 URL 重写是利用 GET 的方法, 在 URL 中加入一些额外的参数来达到 Session Tracking 的目的。URL 改写最大的特征在于可以将 URL 写成额外路径信息、参数或以自定义的格式来表达所要传达的信息。但是, URL 在长度上有一定的限制, 因此在使用上须要考虑长度的大小。

URL 重写的方式，以额外路径信息表达时，URL 看起来如下：

```
http://www.jsptw.com.tw/tracking/userid=S585/  
http://www.jsptw.com.tw/tracking.jsp?userid=S585&password=1234
```

不过它最大的缺点就是：这些数据暴露在浏览器上，因此这对安全性有很大的疑虑。

10-2-4 Cookie

Cookie 对 web 应用程序在 Session Tracking 上有很大的助益。所谓的 Cookie (中文翻译为小饼干)是一个小小的文本文件，它是以 key、value 的方式将 Session Tracking 的内容记录在这个文本文件内，这个文本文件通常存在于你的浏览器的暂存区内。

Cookie 的运作原理与前面两者不尽相同，Cookie 是 Server 利用 HTTP 响应的表头传送一段信息给用户的浏览器，然后浏览器再将表头上有关 Cookie 的信息建成一个文本文件，并且把 key 和 value 的值储存起来，等到浏览器每次存取到这台 Server 时，Cookie 便会随着请求传送给 Server，让 Server 读取 Cookie 的信息来辨识这个用户。

但是这又引发了一个很大的问题，假若你在网络咖啡店连上 amazon 买了一本书，然后你离开了这家网络咖啡店，此时你的 Cookie 还存在浏览器的暂存区内。此时，另一个使用这台计算机的人因为你的 Cookie 存在此浏览器内，而使得 amazon 的 Server 误认为是你还在 amazon 上，因而一时好玩在 amazon 内替你买了一百本书，那岂不是很糟糕？不过每个 Cookie 都可以设定有效时间，如果 Cookie 的有效期限已经过了，则浏览器将不会再把 Cookie 随着请求送出去，而 Cookie 有效时间的长短则由程序员视情况而定。

在 Java 的 Servlet API 中提供 `javax.servlet.http.Cookie` 类来让程序员处理 Cookie。以下是利用 Servlet API 提供的 Cookie 所写成的投票范例程序，Cookie 在这里最主要的用处在于避免同一个人在十分钟之内重复投票。不过，如果真的要做到一人只能投一票的功能，应该必须有身份确认的机制，才够达成目标。

Cookie 的使用上仍然有一些问题有待解决，例如：Cookie 最为人所争议的便是它在隐私权上的问题，因为 Cookie 强大且持续性高，因此它能长时间地追踪用户，甚至是单一用户的每个请求，这种强大的追踪能力使得网站能够在用户无所察觉之下，观察用户的一举一动。然而有些人并不乐意将自己浏览的行为，成为网站用来赚钱的工具，因此可能选择不接受 Cookie，一旦如此，便无法利用 Cookie 来达到 Session Tracking 的功能。再者，有些老旧的浏览器并不提供 Cookie 的机制，也同样使 Cookie 英雄无用武之地。

10-2-5 使用 Servlet 的 Session API

在现代技术中，传统的会话追踪方式已经稍嫌落伍，不合时宜。在 Servlet 发展出来的时候，就已经包含着会话的对象 session。这强大的功能使得 JSP 一诞生便具有会话追踪 Client 的能力，这个会话的对象 session，事实上是 `javax.servlet.http.HttpSession` 的一个参考对象。支持 Servlet 的 Web Server 利用上面探讨过的两种机制：Cookie 和 URL 重写来实现 session。

经由传送一个 session 的 ID 到 Client 端的 Cookie 上,接着判别 Client 的请求中所夹带过来的 Cookie 信息,来比对发出请求的用户是否为同一人。当 Client 端不接受 Cookie 的时候,可以利用 URL 重写的方式将 URL 编码,把 session ID 编码在所欲产生页面之 URL 上,此时须利用 response 对象内的 encodeURL()或 encodeRedirectURL()方法,来将 URL 做一个编码的动作。举例来说:

```
<a href="/CH7/HelloJSTL">Click here</a>
```

应该改为:

```
<a href='%<%= response.encodeURL("/CH7/HelloJSTL.jsp") %>'>Click here</a>
```

假若你须要使用重新导向 URL 时,例如:

```
response.sendRedirect("/CH7/HelloJSTL.jsp");
```

也应该改为:

```
response.sendRedirect(response.encodeRedirectUrl("/CH7/HelloJSTL.jsp"));
```

注意

将 session 的 ID 以 URL 的编码方式进行时,需将每一页都编码,才能保留住 session 的 ID。如果遇到没有编码的 URL,则无法进行 Session Tracking。

下面是一个记录同一用户到站次数的计数器。

■ SessionCounter.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<html>
<head>
  <title>CH10 - SessionCounter.jsp</title>
</head>
<body>

<h2>记录用户到站次数</h2>

<c:set var="time" value="${sessionScope.time}" />

<c:choose>
  <c:when test="${ time != NULL }">
    <c:set var="time" value="${time + 1}" scope="session" />
    欢迎光临本站<br>
    你是第 ${sessionScope.time} 次光临本站<br>
  </c:when>
  <c:otherwise>
    <c:set var="time" value="1" scope="session" />
    欢迎光临本站<br>
  </c:otherwise>
</c:choose>
```

JSP2.0 技术手册

```

        你是第 ${sessionScope.time} 次光临本站<br>
    </c:otherwise>
</c:choose>

</body>
</html>

```

SessionCounter.jsp 记录单一用户每次光临的次数。首先, 从 *sessionScope* 内取出 *time* 的值, 如果值不为 NULL 时, 就累加 1, 再存回 *time* 内; 如果取出来的值为 NULL 时, 则代表用户是第一次光临此网页, 因此把 *time* 的值设为 1。 *SessionCounter.jsp* 的执行结果如图 10-3。



图 10-3 *SessionCounter.jsp* 的执行结果

10-3 Session 的生命周期

session 与 *Cookie* 一样拥有特定的生命周期。一个 *session* 可以利用 *isNew()* 方法来得知是否为一个新的 *session*。所谓“新”的 *session* 意思是说: 它已被 *Server* 产生, 但是 *Client* 尚未被告知。

一般来说: *session* 在一段时间没有作用就会自动失效, 也就是 *Server* 会自动控管 *session* 失效的时间, 不过时间的长短通常由 *JSP Container* 而异, 每个 *JSP Container* 所设定 *session* 失效的时间各有所不同, 但是当设计者有特别需求时, 也可以自己手动设定 *session* 过期的时间。以 *Tomcat 5.0* 为例, 你可以在 *web.xml* 中设定 *session* 的过期时间为 30 分钟, 如下:

```

■ web.xml
<web-app>
.....
<session-config>
    <session-timeout>30</session-timeout>
</session-config>
.....
</web-app>

```

除此之外, 我们也可以利用 *HttpSession* 的 *setMaxInactiveInterval()* 方法来设定 *session* 在一定时间内没有作用就过期, *Server* 就会将 *session* 的资源释放掉。另外也可以使用 *HttpSession* 类的 *invalidate()* 方法, *session* 不论设定多久的时间为过期, 将立即失效。下面范例我们示范

手动控制 session 的过期时间，以节省 Server 资源的方法。

■ SessionLife.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>

<html>
<head>
  <title>CH10 - SessionLife.jsp</title>
</head>
<body>

<h2>Session 生命周期范例</h2>

<%
  // 如果 session 是新的，设定 session 的初值
  if(session.isNew())
  {
    // 设定 session 若十秒内没有活动则使 Session 过期
    session.setMaxInactiveInterval(10);

    // 将此 Session time out 的秒数加入过期时间中
    session.setAttribute("expire", "10");
    out.println("设定 Session 若十秒内没有活动则使 Session 过期");
  }
  else
  {
    String str_expire_time = (String)session.getAttribute("expire");

    // 取得 session 建构的时间
    long create_time = session.getCreationTime();
    long access_time = session.getLastAccessedTime();
    long current_time = System.currentTimeMillis();

    long exist_time = (current_time - create_time) / 1000;
    out.println("session 已存在 "+ exist_time + "秒");

    // 如果 session 存在的时间超过 30 秒，则将 session 删除
    if (exist_time >= 30)
    {
      out.println("session 时间已到...自动失效");
      session.invalidate();
    }
  }
%>

</body>
</html>
```

SessionLife.jsp 范例中，我们使用一些设定 session 的生命周期所需的方法，如 `isNew()`、`invalidate()`、`getCreationTime()`、`getLastAccessedTime()` 和 `setMaxInactiveInterval()`。

我们先判定 session 是否为新的 session，如果是新的 session，设定 session 多久不活动则失效，本范例是设为 10 秒。

```
session.setMaxInactiveInterval(10);
```

注意

在 web.xml 中，`<session-timeout>30</session-timeout>` 的单位是 分钟，即 30 分钟；`session.setMaxInactiveInterval(30)` 的单位是 秒，即 30 秒。

然后利用手动方式计算 session 存在 Server 的时间，如果超过 30 秒则失效，如下面这段程序所示：

```
long create_time = session.getCreationTime();
long access_time = session.getLastAccessedTime();
long current_time = System.currentTimeMillis();
long exist_time = (current_time - create_time) / 1000;

// 如果 session 存在的时间超过 30 秒，则将 session 移除
if (exist_time >= 30)
{
    out.println("<h1>session 时间已到...自动失效</h1>");
    session.invalidate();
}
```

最后 *SessionLife.jsp* 一开始执行如图 10-4，若开始执行后十秒之内，不再执行时，session 将自动移除。当过了 30 秒后，再次执行 *SessionLife.jsp* 时，如图 10-5 所示，session 自动被移除。



图 10-4 SessionLife.jsp 的执行结果



图 10-5 经过断断续续的执行，session 存在 30 秒之后，自动被移除

10-4 HttpSessionBindingListener 接口

当你在利用 session 的时候，有时候会遇到一些情形，如：需要在 session 过期时，将 session 数据写回数据库或文件，或者将某个负责记录用户所点击页面数据的对象加入 session 中。此时，你可能就需要实现 `javax.servlet.http.HttpSessionBindingListener` 接口，此接口有两个必须实现的方法：

JSP2.0 技术手册

```
public void valueBound(HttpSessionBindingEvent event)
public void valueUnbound(HttpSessionBindingEvent event)
```

任何实现 javax.servlet.http.HttpSessionBindingListener 接口的对象, 当该对象加入 session 时, 会自动调用 valueBound() 方法; 而当 session 移除时, 则也会自动调用 valueUnbound()。详细实现方法请看下面范例。

这个范例主要是: 列出现在目前的用户名单, 假若有用户登录进来, 它就会出现在在线用户名单上, 但是如果这个用户停止活动超过一段时间时(范例默认 10 秒钟), 将从在线用户名单上移除。此范例包括两个类: UserList.java、UserTrace.java 和 Login.html、UserList.jsp。

先介绍 UserList 用户名单类, 用来储存在线有多少用户。

■ UserList.java

```
package tw.com.javaworld.CH10;

import java.util.*;

public class UserList {
    private Vector container;

    // UserList 算是一个容器, 整个应用程序内只有一个用户列表类
    private static UserList instance = new UserList();

    // 以 private 的方式调用构造函数, 避免被外界产生新的 instance
    private UserList() {
        container = new Vector();
    }

    // 供外界使用的 instance
    public static UserList getInstance() {
        return instance;
    }

    // 新增用户到用户列表内
    public void addUser(String user) {
        if (user != null) {
            container.addElement(user);
        }
    }

    // 列出所有在线用户
    public Enumeration getList() {
        return container.elements();
    }

    // 移除已离线的用户
    public void removeUser(String user) {
        if (user != null) {
```

JSP2.0 技术手册

```

        container.removeElement(user);
    }
}

```

第二个为记录用户数据的类对象(UserTrace), 主要用来追踪用户所做的事情:

■ UserTrace.java

```

package tw.com.javaworld.CH10;

import java.util.*;
import javax.servlet.http.*;
import tw.com.javaworld.CH10.UserList;

public class UserTrace implements
    javax.servlet.http.HttpSessionBindingListener {

    private String user_name;
    private UserList container = UserList.getInstance();

    public String getUserName() {
        return user_name;
    }

    public void setUserName(String name) {
        user_name = name;
    }

    // 当 UserTrace 被加入 session 对象时会调用此方法
    public void valueBound(HttpSessionBindingEvent event) {
    }

    // 当 UserTrace 被移出 session 对象时会调用此方法
    public void valueUnbound(HttpSessionBindingEvent event) {
        container.removeUser(user_name);
    }
}

```

UserTrace.java 类是实现 javax.servlet.http.HttpSessionBindingListener 接口的, 并且使用到:

```

valueBound(HttpSessionBindingEvent event)
valueUnbound(HttpSessionBindingEvent event)

```

当 UserTrace 对象被加入或是移出 session 对象时, 就会自动调用到这两个方法。此范例希望当用户在一定时间没有活动时, 就移除用户的 session, 并且将用户的代号从在线名单列表上移除。

```

// 当 UserTrace 被移出 session 时会调用此方法
public void valueUnbound(HttpSessionBindingEvent event)
{
    container.removeUser(user_name);
}

```

■ Login.html

```
<html>
<head>
  <title>CH10 - Login.html</title>
<meta http-equiv="Content-Type" content="text/html; charset=GB2312">
</head>
<body>

<h2>登录系统</h2>

<form method="POST" action="UserList.jsp">

<p>请输入用户 ID: <input type="text" name="user_id"></p>
<p><input type="submit" value="登录"></p>

</form>

</body>
</html>
```

Login.html 的执行结果如图 10-6。

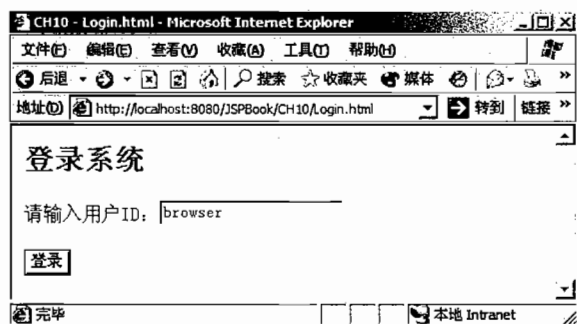


图 10-6 Login.html 的执行结果

■ UserList.jsp

```
<%@ page contentType="text/html;GB2312" %>
<%@ page import="java.util.*" %>
<%@ page import="tw.com.javaworld.CH10.*" %>

<%!
    // 由于用户名单是共享对象且设定为单一 instance
    // 故声明方式不能用<jsp:useBean>的 tag

    userList userList = userList.getInstance();
%>
<jsp:useBean id="usertrace" class="tw.com.javaworld.CH10.UserTrace"
    scope="session"/>
<%
    String user_id = request.getParameter("user_id");
```

JSP2.0 技术手册

```
// 设定用户 id 进入追踪对象中
usertrace.setUserName(user_id);

// 将用户追踪对象加入 session 内
session.setAttribute("usertrace",usertrace);

// 将用户加入用户名单中
userlist.addUser(usertrace.getUserName());

// 设定 session 在 10 秒钟之内没有活动就使 session 无效
session.setMaxInactiveInterval(10);

%>

<html>
<head>
  <meta http-equiv="refresh" content=60>
  <title>CH10 - UserList.jsp</title>
</head>
<body>

<h2>在线用户名单</h2>

<center>
<p>
<textarea rows="9" cols="15">
<%
    // 将在线用户名单打印出来
    Enumeration elements = userlist.getList();

    while(elements.hasMoreElements())
    {
        String name = (String)elements.nextElement();
        out.println(name);
    }
%>
</textarea>
</p>
</center>
</body>
</html>
```

上面的范例用到了两个类：一个是追踪用户数据的类(UserTrace)；另一个则是在线用户名单的类(UserList)，专门记录在线的用户。在这个例子中，我们仅使用到 `valueUnbound()` 方法，用意是当 session 过期时，可以利用 `valueUnbound()` 将位于用户列表内的用户移除，因为当 session 过期时，意味着用户已离线了。

另外我们注意到 UserList 类的写法有点特别，这是因为我们只让类产生一个 instance，让

所有 JSP 可参考到同一个 instance 之中。在实现中, 我们保护构造函数, 使得其他的类对象无法调用此对象的构造函数。

```
private UserList() { container = new Vector(); }
```

并且构建一个惟一的 instance, 将它储存在一个私有的静态变量之中。

```
private static UserList instance = new UserList();
```

再开放一个静态方法供大家存取此 instance。

```
public static UserList getInstance() { return instance; }
```

不过我们在 *UserList.jsp* 中无法直接使用 `<jsp:useBean>` 的标签来产生 *UserList* 类, 只能以 Scriptlet 的方式取得 *UserList* 的 instance 来使用。在 *UserList.jsp* 页面中, 我们可以清楚地看到用户名单只有加入的动作, 并没有将用户移除, 因为这个部分我们是仰赖着 session 的 `valueUnbound()` 方法来判断用户离线与否。

UserList.jsp 的执行结果画面如图 10-7 所示。现在你应该已经清楚的了解到 `javax.servlet.http.HttpSessionBindingListener` 的使用方法了。

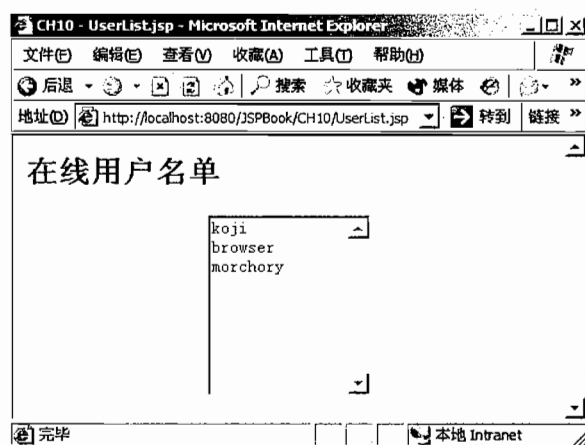


图 10-7 UserList.jsp 的执行结果

10-5 Shopping Cart 范例程序一

这个章节中, 我们将实现 Session Tracking 中最常见的一个例子: 购物车的范例, 使用 session 对象来储存用户所购买的东西。

首先, 我们先构建一个购买商品的 HTML 页面来让用户加入所欲购买的物品, 通常商品区所展示的物品都是从数据库内取得的, 但现在我不想混淆主题, 故先提供一个固定且简单的页面来供参考。

■ Shopping.html

```
<html>
<head>
  <title>CH10 - Shopping.html</title>
<meta http-equiv="Content-Type" content="text/html; charset=GB2312">
</head>
<body>

<h2>即时战略购物区</h2>
<div align="center">
  <center>
    <form action="ShowCartItems.jsp" method="post">
      <table border="1" width="354" height="142">
        <tr>
          <td width="43" height="48">
            <input type="checkbox" name="items" value="AOM">
          </td>
          <td width="355" height="48">AOM</td>
        </tr>
        <tr>
          <td width="43" height="38">
            <input type="checkbox" name="items" value="AOK">
          </td>
          <td width="355" height="38">AOK</td>
        </tr>
        <tr>
          <td width="43" height="36">
            <input type="checkbox" name="items" value="AOC">
          </td>
          <td width="355" height="36">AOC</td>
        </tr>
        <tr>
          <td width="43" height="41">
            <input type="checkbox" name="items" value="Command">
          </td>
          <td width="355" height="41">C&C 2</td>
        </tr>
        <tr>
          <td width="43" height="36">
            <input type="checkbox" name="items" value="SC">
          </td>
          <td width="355" height="36">SC</td>
        </tr>
        <tr>
          <td width="398" height="36" colspan="2">
            <input type="submit" value="加入购物车中">
            <a href="ShowCartItems.jsp?type=show">观看购物车内项目</a>
          </td>
        </tr>
      </table>
    </form>
  </center>
</div>
```

JSP2.0 技术手册

```

</div>

</body>
</html>

```

当用户选择好欲购买的商品后，按下【加入购物车】的按钮后，就会将窗体数据传送至 *ShowCartItem.jsp* 做运算处理。假若按下【观看购物车内项目】时，会传 *type=show* 的参数值至 *ShowCartItem.jsp*。*Shopping.html* 的执行画面如图 10-8。



图 10-8 Shopping.html 的执行结果

ShowCartItem.jsp 处理由 *Shopping.html* 所传来的窗体数据，将商品的数据存入 session 对象中，当要显示购物车内容时，再从 session 对象中逐一取出显示在页面上。

■ ShowCartItem.jsp

```

<%@ page language="java" contentType="text/html;GB2312"%>
<%@ page import="java.util.*" %>

<%!
    public void show_items(HttpSession session, JspWriter out) throws
        java.io.IOException
    {
        out.println("<H1>打印出所有购物车内的对象</H1><br>");

        Enumeration item_names = session.getAttributeNames();

        out.println("<p>你已经购买了下列东西: </p>");
        out.println("<table border=\"1\" width=\"196\" height=\"59\">");
        out.println("<tr><td width=\"121\">商品名称</td><td width=\"59\">
            数量</td></tr>");

        while (item_names.hasMoreElements())

```

JSP2.0 技术手册

```

        {
            String name = (String)item_names.nextElement();

            if(name.endsWith("count"))
            {
                Integer count = (Integer)session.getAttribute(name);
                int length = name.length();

                String real_name = name.substring(0,length-5);

                out.println("<tr>");
                out.println("<td width=\"121\">"+real_name+"</td>");
                out.println("<td width=\"59\">"+count+"</td>");
                out.println("</tr>");
            }
        }

        out.println("</table>");
        out.println("<a href=\"Shopping.html\">继续购物</a>");
    }
    %>

<html>
<head>
    <title>CH10 - ShowCartItems.jsp</title>
    <meta http-equiv="Content-Type" content="text/html; charset=GB2312">
</head>
<body>
<center>

<%
    // 假若是第一次来购物的话
    if (session.isNew())
    {
        String show = request.getParameter("type");

        if(show == null)
        {
            String [] items = request.getParameterValues("items");

            if(items != null)
            {
                for(int i=0 ; i<items.length ; i++)
                {
                    String item_name = items[i];
                    session.setAttribute(item_name,item_name);
                    session.setAttribute(item_name+"count",new
                        Integer(1));
                }

                // 显示购物车目前的内容
                show_items(session,out);
            }
        }
    }
    %>

```

JSP2.0 技术手册

```
// 如果没选择物品时
else
{
    out.println("你未选择加入购物车内之物品 请回上一页<br>");
    out.println("<a href=\"Shopping.html\">回上一页</a>");
}
}
else
{
    out.println("尚未有任何物品加入你的购物车中");
}
}

// 若不是第一次来购物时, 即 session.isNew()为 false
else
{
    String show = request.getParameter("type");

    if(show == null)
    {
        String [] items = request.getParameterValues("items");

        if (items != null)
        {
            for(int i=0 ; i<items.length ; i++)
            {
                String item_name = items[i];
                Integer count = (Integer)session.getAttribute(
                    item_name+"count");

                if(count == null)
                {
                    session.setAttribute(item_name,item_name);
                    session.setAttribute(item_name+"count",
                        new Integer(1));
                }
                else
                {
                    // 将物品计数器做累加的动操作
                    count = new Integer(count.intValue()+1);
                }
                session.setAttribute(item_name+"count",count);
            }

            show_items(session,out);
        }
        else
        {
            out.println("你未选择加入购物车内的物品 请回上一页<br>");
            out.println("<a href=\"Shopping.html\">回上一页</a>");
        }
    }
}
```

```

        else
        {
            show_items(session,out);
        }
    }
%>
</center>
</body>
</html>

```

在这个简单的购物车程序中，我们利用了 session 的特性将用户所选购的物品一个一个放入 session 之中，并计算用户点击物品的次数，计算出用户所要购买物品的数量。

ShowCartItem.jsp 中声明一个专门负责显示购物车内容的 show_items 方法，它有两个传值：一为 HttpSession 类型的 session 对象；二为 JspWriter 类型的 out 对象。

```

public void show_items(HttpSession session , JspWriter out)
        throws java.io.IOException
    {
        out.println("<H1>打印所有购物车内的对象</H1><br>");

        String[] item_names = session.getValueNames();

        out.println("<p>你已经购买了下列东西: </p>");
        out.println("<table border=\"1\" width=\"196\" height=\"59\">");
        out.println("<tr><td width=\"121\">商品名称</td>");
            <td width=\"59\">数量</td></tr>");

        for (int i = 0 ; i < item_names.length ; i++)
        {
            String name = item_names[i];

            if(name.endsWith("count"))
            {
                Integer count = (Integer)session.getValue(name);
                int length = name.length();

                String real_name = name.substring(0,length-5);

                out.println("<tr>");
                out.println("<td width=\"121\">"+real_name+"</td>");
                out.println("<td width=\"59\">"+count+"</td>");
                out.println("</tr>");
            }
        }

        out.println("</table>");
        out.println("<a href=\"Shopping.htm\">继续购物</a>");
    }

```

ShowCartItem.jsp 的执行结果如图 10-9。

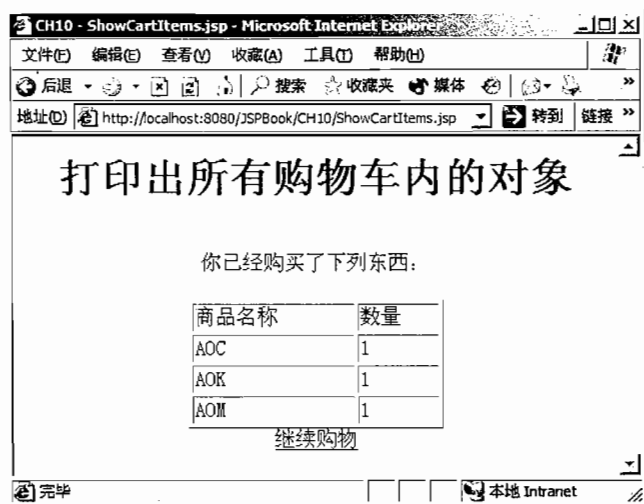


图 10-9 ShowCartItem.jsp 的执行结果

10-6 Shopping Cart 范例程序二

读者看完第一个范例程序后，一定会觉得购物车的功能太过于简陋，只能存放商品名称和购买的数量，因此笔者再举一个功能较完善的购物车范例，不过这个范例的程序代码有些长，读者可能需要先冲泡一杯咖啡，好好静下心来看完这个范例程序。

这个范例最大的不同之处在于：笔者将商品的属性写成一个类对象，例如：这个范例是卖书的在线商店，当然商品就是以书为主。笔者制定每一本书的属性有：书名、作者、出版社、价格等等。因此，写一个 BOOK 的 JavaBean，里面定义一些属性，并且每个属性都有定义设定(setter)和取得(getter)的方法。那么接下来就开始我们这个 Store 的范例程序吧。

这个 Store 的范例程序主要分为四个部分：*Store.html*、*Store.jsp*、*Checkout.jsp* 和 *Book.java*，如表 10-1 所示。

表 10-1

名 称	功能说明
<i>Store.html</i>	显示在线书商所提供贩卖的书籍
<i>Store.jsp</i>	数据的处理，并且显示目前购物车的内容
<i>Checkout.jsp</i>	显示最后购买的物品清单和金额
<i>Book.java</i>	它是一个 JavaBean，内容为书籍的属性：书名、作者、出版商、价格、数量

下面就开始说明这几个程序，首先介绍 *Store.html*，它主要是一个在线商店的商品列单，用户点击欲购买的商品之后，将窗体的数据传送到 *Store.jsp* 做处理。

■ Store.html

```

<html>
<head>
  <title>CH10 - Store.html</title>
  <meta http-equiv="Content-Type" content="text/html; charset=GB2312">
</head>
<body>

<h2>网络书店</h2>
<table border="1" width="631">
  <tr bgcolor="#999999">
    <td width="194"><div align="center"><b>书名</b></div></td>
    <td width="116"><div align="center"><b>作者</b></div></td>
    <td width="78"><div align="center"><b>出版社</b></div></td>
    <td width="47"><div align="center"><b>价格</b></div></td>
    <td width="58"><div align="center"><b>数量</b></div></td>
    <td width="100"><div align="center"><b></b></div></td>
  </tr>
  <form name="shoppingForm" action="Store.jsp" method="POST">
    <tr>
      <td width="174"><div align="center">JSP 2.0 技术手册</div></td>
      <td width="101"><div align="center">林上杰</div></td>
      <td width="57"><div align="center">电子工业出版社</div></td>
      <td width="47"><div align="right">50</div></td>
      <td width="93">数量:
        <input type="text" name="quantity" size="3" value=1>
      </td>
      <td width="119">
        <input type="submit" name="Submit" value="放入购物车">
      </td>
    </tr>
    <input type="hidden" name="name" value="JSP 2.0 技术手册">
    <input type="hidden" name="author" value="林上杰">
    <input type="hidden" name="publisher" value="电子工业出版社">
    <input type="hidden" name="price" value="600">
    <input type="hidden" name="action" value="ADD">
  </form>
  <form name="shoppingForm" action="Store.jsp" method="POST">
    <tr>
      <td width="174"><div align="center">紫牛——让产品自己说故事</div></td>
      <td width="101"><div align="center">赛斯·高汀</div></td>
      <td width="57"><div align="center">商智</div></td>
      <td width="47"><div align="right">237</div></td>
      <td width="93">数量:
        <input type="text" name="quantity" size="3" value=1>
      </td>
      <td width="119">
        <input type="submit" name="Submit" value="放入购物车">
      </td>
    </tr>
  </form>

```

JSP2.0 技术手册

```

</tr>
<input type="hidden" name="name" value="紫牛——让产品自己说故事">
<input type="hidden" name="author" value="赛斯·高汀">
<input type="hidden" name="publisher" value="商智">
<input type="hidden" name="price" value="237">
<input type="hidden" name="action" value="ADD">
</form>
<form name="shoppingForm" action="Store.jsp" method="POST">
<tr>
<td width="174"><div align="center">1421-中国发现世界</div></td>
<td width="101"><div align="center">孟西士</div></td>
<td width="57"><div align="center">远流</div></td>
<td width="47"><div align="right">470</div></td>
<td width="93">数量:
<input type="text" name="quantity" size="3" value="1">
</td>
<td width="119">
<input type="submit" name="Submit" value="放入购物车">
</td>
</tr>
<input type="hidden" name="name" value="1421-中国发现世界">
<input type="hidden" name="author" value="孟西士">
<input type="hidden" name="publisher" value="远流">
<input type="hidden" name="price" value="470">
<input type="hidden" name="action" value="ADD">
</form>
<form name="shoppingForm" action="Store.jsp" method="POST">
<tr>
<td width="174"><div align="center">如何打败可口可乐</div></td>
<td width="101"><div align="center">翟若适</div></td>
<td width="57"><div align="center">联合文学</div></td>
<td width="47"><div align="right">180</div></td>
<td width="93">数量:
<input type="text" name="quantity" size="3" value="1">
</td>
<td width="119">
<input type="submit" name="Submit" value="放入购物车">
</td>
</tr>
<input type="hidden" name="name" value="如何打败可口可乐">
<input type="hidden" name="author" value="翟若适">
<input type="hidden" name="publisher" value="联合文学">
<input type="hidden" name="price" value="180">
<input type="hidden" name="action" value="ADD">
</form>
<form name="shoppingForm" action="Store.jsp" method="POST">
<tr>
<td width="174"><div align="center">菲奥利娜逆势出击</div></td>
<td width="101"><div align="center">彼得·鲍洛斯</div></td>

```

```

<td width="57"><div align="center">商周出版</div></td>
<td width="47"><div align="right">300</div></td>
<td width="93">数量:
  <input type="text" name="quantity" size="3" value="1">
</td>
<td width="119">
  <input type="submit" name="Submit" value="放入购物车">
</td>
</tr>
<input type="hidden" name="name" value="菲奥利娜逆势出击">
<input type="hidden" name="author" value="彼得·鲍洛斯">
<input type="hidden" name="publisher" value="商周出版">
<input type="hidden" name="price" value="300">
<input type="hidden" name="action" value="ADD">
</form>
</table>

</body>
</html>

```

Store.html 的执行结果如图 10-10 所示。

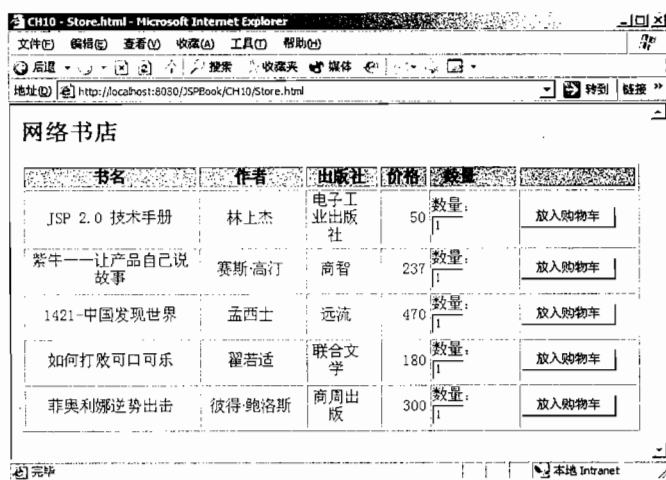


图 10-10 Store.html 的执行结果

再来介绍 Store 范例的主要程序：*Store.jsp*，它接收从 *Store.html* 传来的窗体数据，处理所有和购物车有关的数据，如：新增物品至购物车中、删除购物车中的物品等等。

■ Store.jsp

```

<%@ page import="java.io.*,java.util.*" %>
<%@ page import="tw.com.javaworld.CH10.Book" %>
<%
  Vector buylist = (Vector)session.getAttribute("shoppingcart");
  String action = request.getParameter("action");

```

JSP2.0 技术手册

```

// 删除购物车中的书籍
if (action.equals("DELETE"))
{
    String del = request.getParameter("del");
    int d = (new Integer(del)).intValue();
    buylist.removeElementAt(d);
}

// 新增书籍至购物车中
else if (action.equals("ADD"))
{
    boolean match=false;
%>
<jsp:useBean id="newBook" scope="page" class=
    "tw.com.javaworld.CH10.Book"/>
<jsp:setProperty name="newBook" property="*" />
<%
    //新增第一本书籍至购物车时
    if (buylist == null)
    {
        buylist = new Vector();
        buylist.addElement(newBook);
    }
    else
    {
        for (int i=0; i< buylist.size(); i++)
        {
            Book book = (Book) buylist.elementAt(i);

            // 假若新增的书籍和购物车的书籍一样时
            if(book.getName().equals(newBook.getName()))
            {
                book.setQuantity(book.getQuantity()+
                    newBook.getQuantity());
                buylist.setElementAt(book,i);
                match = true;
            } //end of if name matches
        } // end of for

        // 假若新增的书籍和购物车的书籍不一样时
        if (!match)
            buylist.addElement(newBook);
    }

    session.setAttribute("shoppingcart", buylist);

    if (buylist != null && (buylist.size() > 0))
    {
%>
<html>
<head>

```



```

<title>CH10 - Store.jsp</title>
</head>
<body>

<h2>目前您购物车的内容如下: </h2>

<table border="1" width="631">
  <tr bgcolor="#999999">
    <td width="194"><div align="center"><b>书名</b></div></td>
    <td width="81"><div align="center"><b>作者</b></div></td>
    <td width="93"><div align="center"><b>出版社</b></div></td>
    <td width="57"><div align="center"><b>价格</b></div></td>
    <td width="47"><div align="center"><b>数量</b></div></td>
    <td width="119"><div align="center"><b></b></div></td>
  </tr>
  <%
    for (int index=0; index < buylist.size();index++)
    {
      Book order = (Book)buylist.elementAt(index);
    %>
    <tr>
      <td><b><%= order.getName() %></b></td>
      <td><b><%= order.getAuthor() %></b></td>
      <td><b><%= order.getPublisher() %></b></td>
      <td><b><div align="right"><%= order.getPrice() %></div></b></td>
      <td><b><div align="right"><%= order.getQuantity() %></div></b></td>
      <td>
        <form name="deleteForm" action="Store.jsp" method="POST">
          <input type="submit" value="Delete">
          <input type="hidden" name="del" value='<%= index %>'>
          <input type="hidden" name="action" value="DELETE">
        </form>
      </td>
    </tr>
  <%
    }
  %>
</table>
<p>
  <a href="Store.html">继续购物</a>
  <form name="checkoutForm" action="Checkout.jsp" method="POST">
    <input type="hidden" name="action" value="CHECKOUT">
    <input type="submit" name="Checkout" value="付款结账">
  </form>
  <%
    }
  else
  {
  %>
    <h2>目前您的购物车没有任何物品: </h2><br>
    <a href="Store.html">继续购物</a>

```



```

<%
    }
%>

</body>
</html>

```

Store.jsp 中, 笔者主要利用 *Vector* 来完成购物车的功能, 将代表每一本书的 *Book* 对象都加入至一个 *Vector* 数组中, 如下所示:

```

<jsp:useBean id="newBook" scope="page"
class="tw.com.javaworld.CH10.Book"/>
<jsp:setProperty name="newBook" property="*" />
.....
buylist = new Vector();
buylist.addElement(newBook);

```

然后再将整个 *Vector* 对象放入 *session* 对象中, 让购物车除了能放入许多的商品之外, 并且还能够让服务器辨认清楚每一个购物车是属于哪一个用户的。

Store.jsp 分别处理删除物品和新增物品的两项功能。我们先来看删除物品, 假若前端传来的信息是请求执行删除购物车中物品时, *Store.jsp* 会接收到两个参数: *action* 和 *del*。 *action* 是用来告诉服务器, 现在用户请求什么服务, 是新增还是删除, 若 *action* 的值为 *DELETE* 时, 表示执行的是删除操作。确认是删除时, *Store.jsp* 还会接收到 *del* 的值, 它是指: 要删除的项目顺序号码, 如下所示:

```

String action = request.getParameter("action");

if (action.equals("DELETE"))
{
    String del = request.getParameter("del");
    int d = (new Integer(del)).intValue();
    buylist.removeElementAt(d);
}

```

另一方面, 若是执行新增物品时, 我们先要做的事情就是把用户选择购买的商品属性存入 *JavaBean* 中, 笔者是利用 *<jsp:useBean>* 和 *<jsp:setProperty>* 两个标签来完成这项工作。

```

<jsp:useBean id="newBook" scope="page" class="mysession.BOOK"/>
<jsp:setProperty name="newBook" property="*" />

```

新增物品还有几个需要特别小心的地方, 例如: 若是第一次新增物品, 我们就必须产生一个新的 *Vector*, 以供用户使用。如下列这段程序:

```

// 新增第一本书籍至购物车时
if (buylist == null)
{
    buylist = new Vector();
    buylist.addElement(newBook);
}

```

新增部分最麻烦的地方就是在新增物品时, 若购物车已经有那一项商品时, 必须将原来商

品的数量加上后来新增的数量。如下：

```
for (int i=0; i< buylist.size(); i++)
{
    Book book = (Book) buylist.elementAt(i);

    // 假若新增的书籍和购物车的书籍一样时
    if (book.getName().equals(newBook.getName()))
    {
        book.setQuantity(book.getQuantity()+newBook.getQuantity());
        buylist.setElementAt(book,i);
        match = true;
    }
}
```

最后显示目前购物车的商品清单，如图 10-11 所示。

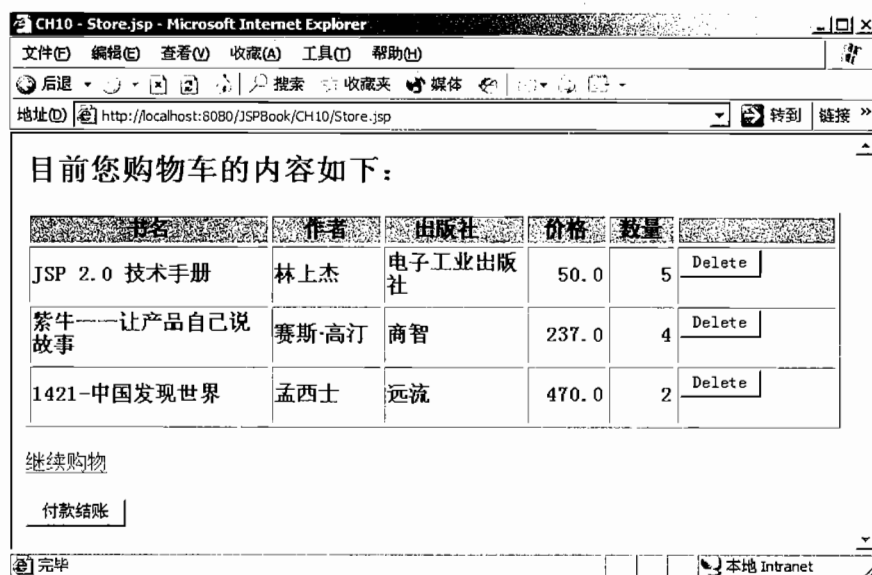


图 10-11 Store.jsp 的执行结果

网络书店范例用到的 JavaBean: **Book.java**, 它的内容都是有关书的属性, 有书名、作者、出版社、价格等等。假若读者觉得这些属性还不够, 只要自行再增加 **Book.java** 的属性, 如: 库存量、折扣价、简介等等。这样的做法可以增加程序功能扩充的弹性, 不须要更动到其他的程序, 日后的维护也比较方便。

■ Book.java

```
package tw.com.javaworld.CH10;
```

```
public class Book {
```

```
    public Book() {
```

JSP2.0 技术手册

```

    name = "";
    author = "";
    publisher = "";
    price = 0;
    quantity = 0;
}

private String name;
private String author;
private String publisher;
private float price;
private int quantity;

public String getName() {
    return name;
}
public String getAuthor() {
    return author;
}
public String getPublisher() {
    return publisher;
}
public void setPrice(float newPrice) {
    price = newPrice;
}
public float getPrice() {
    return price;
}
public void setQuantity(int newQuantity) {
    quantity = newQuantity;
}
public int getQuantity() {
    return quantity;
}
public void setPublisher(String newPublisher) {
    publisher = newPublisher;
}
public void setAuthor(String newAuthor) {
    author = newAuthor;
}
public void setName(String newName) {
    name = newName;
}
}

```

最后就是结账部分 *Checkout.jsp*。它主要负责将购物车的物品列成一份清单，并且计算清楚总共购买的金额。

■ *Checkout.jsp*

```

<%@ page import="java.util.*" %>
<%@ page import="tw.com.javaworld.CH10.Book" %>

```

```

<html>
<head>
  <title>CH10 - Checkout.jsp</title>
</head>
<body>

<h2>网络书店 - 结账</h2>
<center>
  <table border="1" width="631">
    <tr bgcolor="#999999">
      <td width="194"><div align="center"><b>书名</b></div></td>
      <td width="81"><div align="center"><b>作者</b></div></td>
      <td width="57"><div align="center"><b>出版社</b></div></td>
      <td width="93"><div align="center"><b>价格</b></div></td>
      <td width="47"><div align="center"><b>数量</b></div></td>
      <td width="119"><div align="center"><b></b></div></td>
    </tr>
    <%
      Vector buylist = (Vector) session.getAttribute("shoppingcart");
      float total = 0;

      for (int i=0; i < buylist.size(); i++)
      {
        Book order = (Book)buylist.elementAt(i);
        String name = order.getName();
        String author = order.getAuthor();
        String publisher = order.getPublisher();
        float price = order.getPrice();
        int quantity = order.getQuantity();
    %>
    <tr>
      <td><b><%= name %></b></td>
      <td><b><%= author %></b></td>
      <td><b><%= publisher %></b></td>
      <td><b><div align="right"><%= price %></div></b></td>
      <td><b><div align="right"><%= quantity %></div></b></td>
    </tr>
    <%
      total += (price * quantity);
    %>
    <tr>
      <td><div align="right"><b>总金额: </b></div></td>
      <td><div align="right"><b><%= total %></b></div></td>
    </tr>
  </table>
</center>

```

```
</table>
<p>
<a href="Store.html">是否继续购物</a>
</center>

</body>
</html>
```

Checkout.jsp 的执行画面如图 10-12。

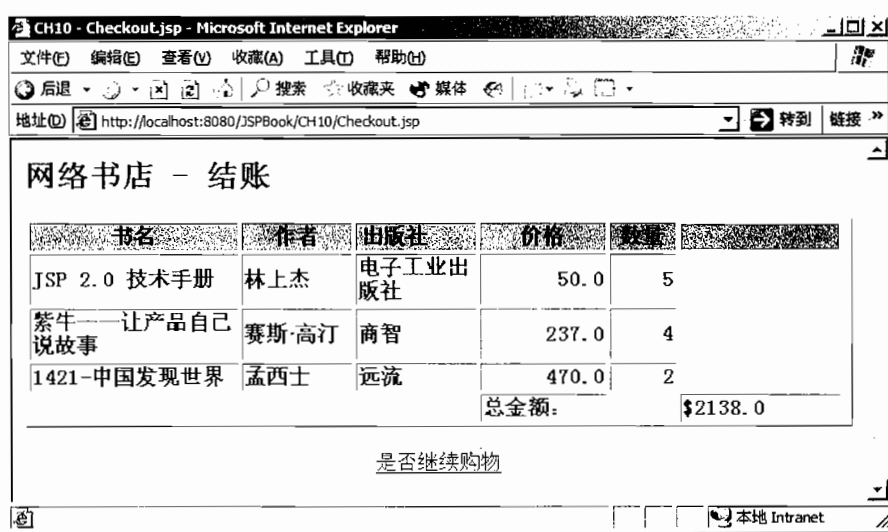


图 10-12 Checkout.jsp 的执行结果

■ 结论

Store 这个范例程序，目前只卖图书，因此也只定义一个书的类。不过一个 Store 应该还会卖其他的商品，但是每一项商品也都有自己专有的属性。例如：今天我想扩充我的在线商店，我还想卖食物，因此我应该可能要再定义一个食物的类：FOOD，其中 FOOD 的属性可能有：食物名称、制造成份、制造日期、制造厂商等等。然后当用户点击想要买的食物，再把它加进 Vector 类型的购物车中就完成了，是不是很简单，而且也不须要更改到其他的程序，这就是对象化的好处之一。

11

第十一章

Filter 与 Listener

Filter 和 Listener 都是在 Servlet 2.3 和 JSP 1.2 才新增加进来的功能。本章将介绍有关 Filter、Listener 的设定和使用方式，并且撰写几个范例程序，让读者更能了解两者的运作方式和使用时机。接下来笔者将分以下 10 节来介绍：

- 11-1 Filter 简介
- 11-2 Filter 的运作方式
- 11-3 实现阶段第一个 Filter
- 11-4 对请求做统一的认证处理
- 11-5 ServletRequest 和 ServletResponse 之 Wrapper 类
- 11-6 使用 Filter 来解决中文问题
- 11-7 Listener 接口简介
- 11-8 ServletContext Listener
- 11-9 HttpSession Listener
- 11-10 ServletRequest Listener

JSP2.0 技术手册

11-1 Filter 简介

什么是 Filter? 依据字面上的中文意思为过滤器。很多时候会利用到它, 例如: 在线游戏、web 聊天室或 E-mail 等等, 可以利用 Filter 来过滤不雅字句或者拒绝对象的信息或信件。Servlet 2.3 的 Filter 运作原理也很类似: 当用户的请求到达指定的网页之前, 可以借助过滤器(filter)来改变这些请求的内容; 同样地, 当执行结果要响应到用户之前, 若先经过过滤器, 就可以修改输出的内容。

在 Servlet 2.3 以前, 若想要有 Filter 的功能, 必须先写一个父类让其他想通过同样处理的 Servlet 都继承该父类; 如果我们每次想改变经过的过滤器, 就得重新改变继承的对象, 且必须重新编译它一次。

Servlet 2.3 之后, 实现阶段 Filter 的过程只须在 *web.xml* 中设定好相关设定即可, 可以让开发者能轻易地加入新的 Filter 机制、修改 Filter 功能, 而不须要更动原来写好的 Servlet、JSP 和其他静态网页。这对已经在运作的服务管理者来说也是一大福音, 到底在哪些情况下有可能会使用到 Filter 呢? 以下是几种 Filter 可行的运用方法:

- 统一的认证处理 (authentication filters)
- 对用户的请求做检查、做更精确的记录 (logging and auditing filters)
- 监视或对用户所传递的参数做前置处理, 例如: 防止数据隐码攻击(SQL Injection) (encryption filters)
- 改变图像文件的格式 (image conversion filters)
- 对响应做编码的动作 (MIME-type chain filters)
- 对响应做压缩处理 (data compression filters)
- 对 XML 的输出使用 XSLT 来转换 (XSLT filters that transform XML content)

11-2 Filter 的运作方式

Filter 的基本运作方式如图 11-1。

从图 11-1 可以发现到 Filter 的运作方式如同一层接一层, 一直从最外面的 Filter 做到最里面的原始网页, 然后再把响应一层一层转送出去, 最后产生回传结果给用户。但是 Filter 也不是只能乖乖照这个顺序转送, 首先 Filter 必须使用 `doFilter()` 才可以继续转送到下一个 Filter, 因此可以选择某种条件下让 Filter 不要调用 `doFilter()`, 而是通过其他方法转向到其他网页、或者是抛出异常处理。Filter 主要可以通过 `RequestDispatcher` 的 `forward()`、`include()` 或 `HttpServletResponse` 的 `sendRedirect()`、`sendError()` 等方法来转向到其他网页的动作。

我们可以在 *web.xml* 中使用 `<dispatcher>` 元素, 用来设定 Filter 所对应的请求方式。它有四种设定, 分别是 `REQUEST`、`FORWARD`、`INCLUDE`、`ERROR`, 以下分别介绍之。

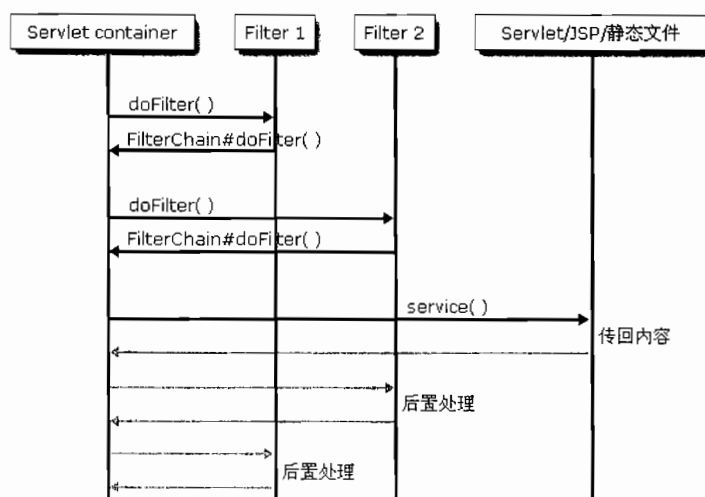


图 11-1 Filter 的运作方式

1. REQUEST

当用户直接对网页做出请求的动作时，才会通过此 Filter。因此当用户通过 RequestDispatcher 方法的转向请求(forward())时，不会通过此 Filter。

2. FORWARD

当用户的请求是通过 RequestDispatcher 的 forward()方法时，才会通过此 Filter。其他请求方式则不会通过此 Filter。

3. INCLUDE

当用户的请求是通过 RequestDispatcher 的 include()方法时，才会通过此 Filter。其他请求方式则不会通过此 Filter。

4. ERROR

当用户的请求是通过错误机制处理的时候，才会通过此 Filter。同样，其他请求方式则不会通过此 Filter。

注意

错误机制处理是指在 web.xml 中，使用<error-page>来处理错误，例如：

```

<error-page>
  <error-code>404</error-code>
  <location>/error404.jsp</location>
</error-page>
<error-page>
  <exception-type>java.lang.Exception</exception-type>
  <location>/except.jsp</location>
</error-page>
  
```

以下简略介绍一下 web.xml 中 Filter 的设定值：

■ web.xml

```

:
<filter>
  <filter-name>filterSample</filter-name>
  <filter-class>twocom.javaworld.CH11.filterSample</filter-class>
</filter>

<filter-mapping>
  <filter-name>filterSample</filter-name>
  <url-pattern>/*</url-pattern>
  <dispatcher>REQUEST</dispatcher>
  <dispatcher>FORWARD</dispatcher>
  <dispatcher>INCLUDE</dispatcher>
  <dispatcher>ERROR</dispatcher>
</filter-mapping>
:

```

Filter 的相关设定有如下几个:

● <filter>

用来声明一个 Filter 的数据

<filter-name>Filter 的名称</filter-name>

定义 Filter 的名称

<filter-class>Filter 的类名称</filter-class>

定义 Filter 的类名称。例如: com.jsptw.filter.filterSample

● <init-param>

init-param 元素包含三个子元素, 分别为 param-name、param-value、description。用来初始化参数。

<param-name>参数名称</param-name>

定义参数名称

<param-value>参数的值</param-value>

指定参数的值

<description>参数说明</description>

参数的说明

● <filter-mapping>

filter-mapping 元素包含两个子元素, filter-name 和 url-pattern。用来定义某一个 Filter 和某些 URL 的对应。

<filter-name>Filter 的名称</filter-name>

定义 Filter 的名称。必须和<filter-name>的名称相同才可以互相对应。

<servlet-name> Servlet 的名称</servlet-name>

定义 Servlet 的名称。

JSP2.0 技术手册

● **<url-pattern>URL</url-pattern>**

Filter 所对应的 URL。

<dispatcher>REQUEST | INCLUDE | FORWARD | ERROR</dispatcher>

设定Filter对应的请求方式，有REQUEST、INCLUDE、FORWARD、ERROR四种，默认为REQUEST。如果设定为INCLUDE，则必须把<url-pattern>改成<servlet-name>，意思为：当RequestDispatcher使用include()的对象为<servlet-name>所设定的对象时，才通过此Filter。接下来看几个相关<dispatcher>设定的例子：

例1:

```
<filter-mapping>
  <filter-name>Logging Filter</filter-name>
  <url-pattern>/products/*</url-pattern>
</filter-mapping>
```

因为没有设定<dispatcher>，所以只有当用户直接对网页做出请求的动作时，才会通过此Filter。因为之前提过，<dispatcher>的默认值为REQUEST。接下来再看一个例子：

例2:

```
<filter-mapping>
  <filter-name>Logging Filter</filter-name>
  <servlet-name>ProductServlet</servlet-name>
  <dispatcher>INCLUDE</dispatcher>
</filter-mapping>
```

上例中<dispatcher>设为INCLUDE，意思是说：当请求从ProductServlet发出且通过RequestDispatcher的include()时，才会通过此Filter。

例3:

```
<filter-mapping>
  <filter-name>Logging Filter</filter-name>
  <url-pattern>/products/*</url-pattern>
  <dispatcher>FORWARD</dispatcher>
  <dispatcher>REQUEST</dispatcher>
</filter-mapping>
```

上例的意思是指：只有在用户直接对路径名称为/products/之下的网页做出请求的动作，或者通过RequestDispatcher的forward()做出转向且路径名称为/products/之下，才会通过此Filter。

11-3 实现阶段第一个 Filter

介绍完 Filter 的基本设定和运作流程之后，笔者在这里举一个 HelloFilter 的例子。

■ **HelloFilter.java**

```
package tw.com.javaworld.CH11;
```

```

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloFilter implements Filter {

    public void init(FilterConfig config) throws ServletException {

    }

    public void doFilter(
        ServletRequest request,
        ServletResponse response,
        FilterChain chain) throws IOException, ServletException {

        System.out.println("Hello Filter !!");
        chain.doFilter(request, response);
    }

    public void destroy() {

    }
}

```

实现阶段一个 Filter 时，都必须实现阶段 javax.servlet.Filter 接口。因此笔者首先介绍 javax.servlet.Filter 这个接口的三个方法。在 Filter 接口中有下列三个方法：

```

public void init(FilterConfig filterConfig)
public void doFilter( ServletRequest request, ServletResponse response,
                    FilterChain chain)
public void destroy( )

```

接下来为读者简单介绍这三个方法：

```

public void init(FilterConfig filterConfig)

```

当 Filter 被加载时，首先执行 init(FilterConfig filterConfig)方法。通常我们会在这里面做初始化的动作。

```

public void doFilter( ServletRequest request, ServletResponse response,
                    FilterChain chain)

```

Filter 处理过程的方法。doFilter 有三个传入的参数值，前两个为传给 Servlet 或 JSP 的 request、response 对象，最后一个参数 chain，则是用来把 request 和 response 对象交给下一个 Filter 的 FilterChain 对象。FilterChain 是用 doFilter()方法来调用下一个 Filter，或者当没有 Filter 需要再被调用时，则调用原始的 Servlet 等网页部分。HelloFilter 中 doFilter()的内容如下：

```

public void doFilter(ServletRequest request,
                    ServletResponse response,
                    FilterChain chain) throws IOException, ServletException {

    System.out.println("Hello Filter !!");
    chain.doFilter(request, response);
}

```

HelloFilter 的 doFilter()并没有对 request 和 response 做任何事，就把 request 和 response

传给下一个 Filter 或其他 Servlet 等等。

```
public void destroy()
```

当 Filter 的生命周期结束时, 会自动调用这个方法, 但是在大多数情况下应该不会用到。

另外我们必须去修改 `web.xml` 文件, HelloFilter 才能被加载至 JSP 容器中。我们在 `web.xml` 中的 `<web-app>` 元素里, 新增下面几行:

```
:
<filter>
  <filter-name>Hello</filter-name>
  <filter-class>tw.com.javaworld.CH11.HelloFilter</filter-class>
</filter>

<filter-mapping>
  <filter-name>Hello</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
:
```

在 `web.xml` 中, 主要设定的就是: 当 Filter 被加载时, 那些网页在接收到请求时, 必须先通过此 Filter。首先必须先定义 Filter 的名称和 Filter 的类, 然后设定对应的网页。

范例中把 `tw.com.javaworld.CH11.HelloFilter` 的名称设定为 `Hello`, 然后对应的网页设定为 `<url-pattern>/* </url-pattern>`, 表示所有的网页被请求时都会经过此 Filter。另外, 因为在这里没有设定 `<dispatcher>` 元素, 所以只要是从用户端发出的请求才会通过此 Filter。

完成设定之后就可以来测试 HelloFilter 了, 测试的方法很简单, 任意执行 Servlet、JSP 或其他静态网页, 就可以发现什么事情都没有发生。HelloFilter 的 `doFilter()` 只在做在 console 中显示的 “Hello Filter !!”, 然后 Filter 再把 request 和 response 传给我们所指定的网页, 所以读者任意开一个已存在的网页, 可以在 Tomcat 的 console 下看到如下结果 (注意: Tomcat 5 提供的启动快捷方式不会弹出 Tomcat console; 关闭已经启动的 Tomcat, 再通过运行 `{Tomcat_Install}\bin\startup.bat` 来重新启动 Tomcat, 则可以看到 Tomcat console)。

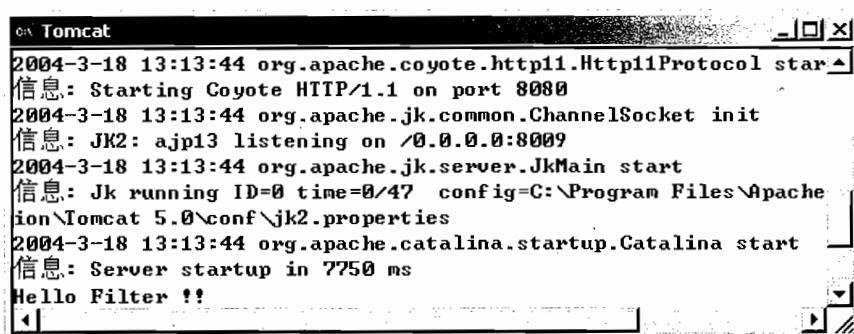


图 11-2 Tomcat Console 所显示的结果

由图 11-2 的最后一行，可以发现到 HelloFilter 确实已经在运作。

11-4 对请求做统一的认证处理

接下来笔者实际写一个范例，这个范例总共有三个文件。分别为 *SessionChecker.java*、*LoginChecker.java* 和 *Login.jsp*。当用户进入到网站时，通过 SessionChecker 检查是否已通过认证，如果他没有经过登录手续，则会把他转向到 *Login.jsp* 中，另外 LoginChecker 还用来检查用户名和密码是否符合。

■ SessionChecker.java

```
package tw.com.javaworld.CH11;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SessionChecker implements Filter {
    private ServletContext context;
    private String targetURI;

    public void init(FilterConfig config) throws ServletException {
        context = config.getServletContext();
        targetURI = config.getInitParameter("targetURI");
    }

    public void doFilter(
        ServletRequest request,
        ServletResponse response,
        FilterChain chain) throws IOException, ServletException {

        HttpServletRequest httpRequest = (HttpServletRequest) request;
        HttpServletResponse httpResponse = (HttpServletResponse) response;
        HttpSession session = httpRequest.getSession(false);

        if (session != null) {
            String passed = (String) session.getAttribute("passed");
            if (passed.equals("true")) {
                chain.doFilter(httpRequest, httpResponse);
                return;
            } else if (passed.equals("passing")) {
                if (new String(httpRequest.getRequestURI()).equals(
                    "/JSPBook/LoginChecker")) {
                    chain.doFilter(httpRequest, httpResponse);
                    return;
                }
            } else { }

            session.removeAttribute("passed");
        }
    }
}
```

JSP2.0 技术手册

```

        StringBuffer requestURL = httpRequest.getRequestURL();
        String query = httpRequest.getQueryString();
        if (query != null)
            requestURL.append(query);
        httpRequest.setAttribute("originalURI", new String(requestURL));

        httpRequest.getRequestDispatcher(targetURI).forward(httpRequest,
            httpResponse);
    }
    public void destroy() {
    }
}

```

SessionChecker.java 中，我们在 *init()* 中调用 *FilterConfig* 的 *getInitParameter()* 来取得在 *web.xml* 中 *targetURI* 的设定值。

```

public void init(FilterConfig config) throws ServletException {
    context = config.getServletContext();
    targetURI = config.getInitParameter("targetURI");
}

```

targetURI 是指如果没有通过认证的用户，将会被转至的网页位置。另外笔者在做完检查的动作后，如果发现用户已经通过认证，则把用户请求转送到原本请求的网页中。如下：

```

if (session != null) {
    String passed = (String) session.getAttribute("passed");
    if (passed.equals("true")) {
        chain.doFilter(httpRequest, httpResponse);
        return;
    } else if (passed.equals("passing")) {
        if (new String(httpRequest.getRequestURI()).equals(
            "/JSPBook/LoginChecker"))
        {
            chain.doFilter(httpRequest, httpResponse);
            return;
        }
    } else {
    }
}
session.removeAttribute("passed");
}

```

检查时分为两种情形：

第一种：当用户已通过身份认证时，直接进入下一个 *Filter* 或原本请求的网页。

第二种：当用户从 *Login.jsp* 进入时，因为用户已打开 *Login.jsp* 这个网页，所以 *session* 中的参数 *passed* 为 *passing*。此时，如果用户直接按下【刷新】，就可以不用密码直接进入请求的网页中，所以必须检查用户是否通过 *Login.jsp* 中的 *Form* 到指定的网页。如果用户按下【刷新】的按钮，则 *getRequestURI()* 值仍为原本请求的网页而不是 *LoginChecker*，所以用户仍会跳回 *Login.jsp* 中。

读者可以发现当 `doFilter()` 执行后，又多加一行 `return`。原因是：Filter 的运作方式是在进入 `doFilter()` 处理完用户原本请求的网页后，会再回到 Filter 中继续往下执行。但是接下来是认证失败的动作才需要处理，所以笔者在此行加上一个 `return`，让成功认证的请求动作直接结束在 `return` 这一行。另外 `session.removeAttribute("passed")` 主要是当发现含有 `passed` 这个参数的值并不正确时，把 `passed` 这个参数去除掉，防止之后认证又读到这个错误的值。

```
StringBuffer requestURL = httpRequest.getRequestURL();
String query = httpRequest.getQueryString();

if (query != null)
    requestURL.append(query);
httpRequest.setAttribute("originalURI", new String(requestURL));
httpRequest.getRequestDispatcher(targetURI).forward(httpRequest,
httpResponse);
```

如果认证失败，将会把用户的请求转置 `Login.jsp` 中，但是在此之前，使用 `httpRequest.getRequestURL()` 和 `httpRequest.getQueryString()` 可以把用户原本指定的目标和传递的值取出来，设在 Attribute 中。当用户认证结束后再把 Attribute 中的值取出来，就可以直接到达之前指定的网页，而不用重新输入之前指定的网页和传递的参数。最后转向到 `Login.jsp` 的方法是调用 `RequestDispatcher` 的 `forward()` 把用户转置过去（见图 11-3）。

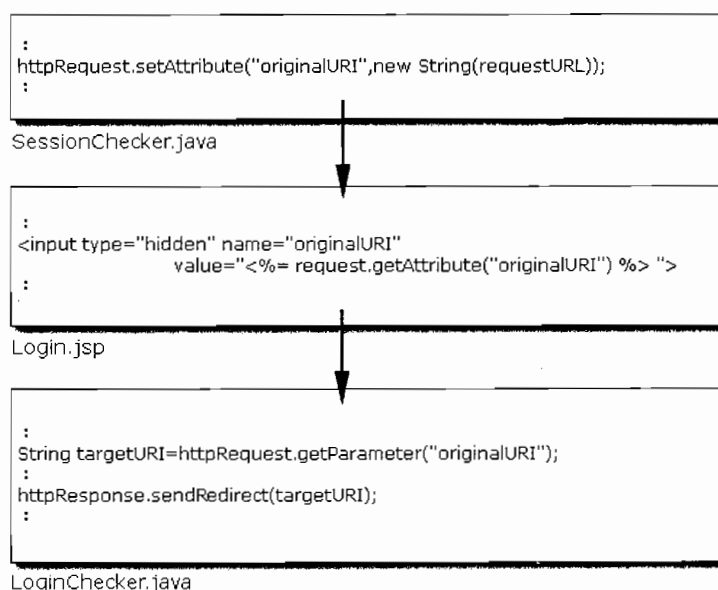


图 11-3 保留用户原本请求的网页技巧

■ Login.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

```

<html>
<head>
  <title>CH11 - Login.jsp</title>
</head>
<body>

<h2>欢迎进入 JSP 2.0 技术手册 范例程序</h2>
<h2>请登录 - 名称: admin 密码: 1234</h2>

<c:set var="passed" value="passing" scope="session" />
<form method="post" action="/JSPBook/LoginChecker">
  <table>
    <tr>
      <th>用户名称: </th>
      <td><input type="text" name="userId"></td>
    </tr>
    <tr>
      <th>用户密码: </th>
      <td><input type="password" name="password"></td>
    </tr>
    <th><input type="hidden" name="originalURI"
value="${requestScope.originalURI}"></th>
    <tr>
      <th><input name="submit" type="submit" value="登录"></th>
    </tr>
  </table>
</form>

</body>
</html>

```

在 *SessionChecker.java* 中,笔者已经将用户原本请求的网址和参数设定至 Request 范围中,所以在 *Login.jsp* 中,笔者把值取出并设定在隐藏字段中,如下所示:

```

<input type="hidden" name="originalURI"
value="${requestScope.originalURI}">

```

此做法最大的用意在于:到 *LoginChecker.java* 通过身份认证时,可以直接从字段得到之前用户请求的网页。在执行时可以从 *Login.jsp* 的源文件中发现, *Login.jsp* 已经储存用户原本请求的网页。如图 11-4 画线处。

■ *LoginChecker.java*

```

package tw.com.javaworld.CH11;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class LoginChecker extends HttpServlet {

    protected void doPost( HttpServletRequest httpRequest,
                          HttpServletResponse httpResponse)

```

JSP2.0 技术手册


```

        throws IOException, ServletException {

    String userId = httpRequest.getParameter("userId");
    String password = httpRequest.getParameter("password");
    String targetURI = httpRequest.getParameter("originalURI");

    if ((!userId.equals("admin")) || (!password.equals("1234"))) {
        throw new ServletException("认证失败");
    }

    HttpSession session = httpRequest.getSession();
    session.setAttribute("passed", "true");
    httpResponse.sendRedirect(targetURI);
}
}

```

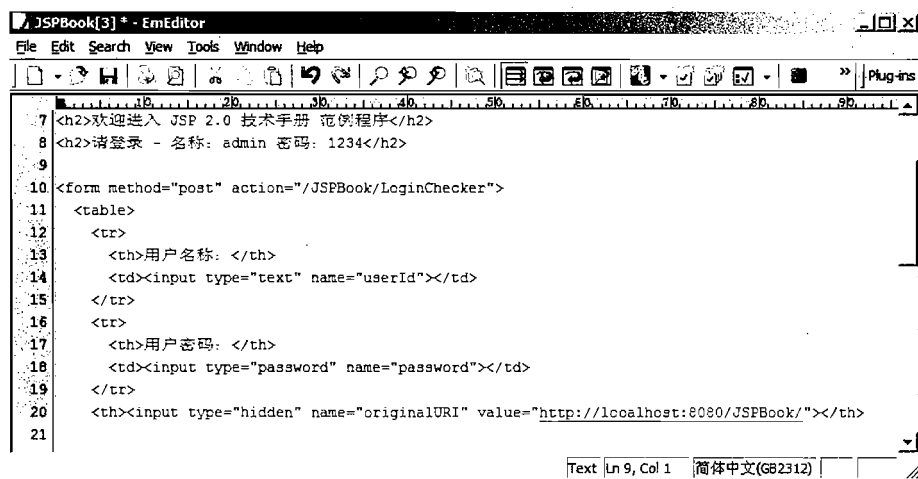


图 11-4 Login.jsp 中隐藏用户最原本请求的网页

在 *LoginChecker.java* 中，笔者检查用户名称和密码是否正确，用户名称为 admin；密码为 1234。如果检查未通过时，将会抛出一个异常，结束动作；如果符合名称和密码，则会在 session 中放入已通过认证的值，即 passed 变量，其值为 true 的动作，如下所示：

```

String targetURI = httpRequest.getParameter("originalURI");

if ((!userId.equals("admin")) || (!password.equals("1234"))) {
    throw new ServletException("认证失败");
}

HttpSession session = httpRequest.getSession();
session.setAttribute("passed", "true");
httpResponse.sendRedirect(targetURI);

```

因为在 *Login.jsp* 中设定了隐藏字段，所以可以直接使用 `getParameter("originalURI")` 来取得用户一开始所要指定的网页和传递参数，接着认证成功后直接转送到用户原本请求的网页。

JSP2.0 技术手册

当我们把以上的三个文件写好后, 接下来就是设定 `web.xml`。我们必须在使用的 `webapp` 下的 `WEB-INF` 中的 `web.xml` 里加入这一段设定值:

■ `web.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<filter>
  <filter-name>SessionChecker</filter-name>
  <filter-class>tw.com.javaworld.CH11.SessionChecker</filter-class>
  <init-param>
    <param-name>targetURI</param-name>
    <param-value>/CH11/Login.jsp</param-value>
  </init-param>
</filter>

<filter-mapping>
  <filter-name>SessionChecker</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>

<servlet>
  <servlet-name>LoginChecker</servlet-name>
  <servlet-class>tw.com.javaworld.CH11.LoginChecker</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>LoginChecker</servlet-name>
  <url-pattern>/LoginChecker</url-pattern>
</servlet-mapping>
:
```

由 `<url-pattern>/*</url-pattern>` 可以看出来, 笔者设定在 JSPBook 站台下的所有网页都必须先通过 SessionChecker。

第一次执行 JSPBook 站台时, 用户一定会被转向至认证画面中, 如图 11-5 所示:

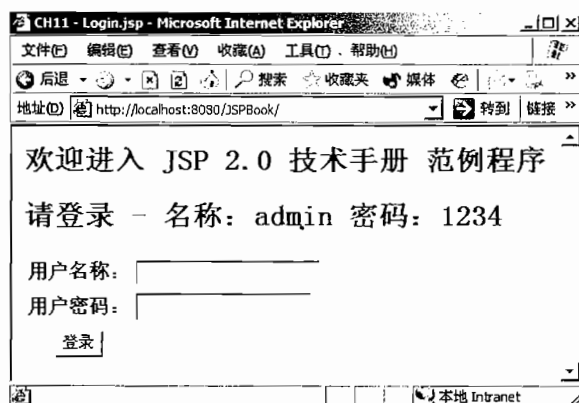


图 11-5 执行本书范例时, 皆需要做登录动作

注意

本书范例的 web.xml 中, SessionChecker 的 <filter-mapping> 设定被当做批注, 所以 SessionChecker 不会执行, 即:

```
:
<!--
<filter-mapping>
  <filter-name>SessionChecker</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
-->
:
```

因此若要有上述的结果, 必须将 <!-- 和 --> 删除掉。

11-5 ServletRequest 和 ServletResponse 之 Wrapper 类

在上一节里, 笔者只对用户的请求做出检查和转向的动作, 如果想改变请求或响应的内容时, 只靠 Filter 是不够用的。例如: 当我们想改变请求的表头时, 会希望 HttpServletRequest 提供如同 HttpServletResponse 的 setHeader()。但这是不可能的事情, 因为 HttpServletRequest 是代表用户端浏览器送来的请求, 而 Servlet 并没有提供接口让我们去修改它。因此为了能够修改它, 必须传入一个改变好表头的 ServletRequest 对象到 Filter 的 doFilter() 中。

我们不想重新制造一个 ServletRequest 出来, 因为我们并没有改变 ServletRequest 的其他部分, 所以为了保持其他和原本 ServletRequest 相同的部分, 必须产生一个重新包装 ServletRequest 的对象, 使其能修改想修改的部分, 并保持其他未修改的原本部分。

为了达到以上功能, 在 Servlet 2.4 规范中提供对于 Filter 机制非常重要的 ServletRequest 和 ServletResponse 之 Wrapper 类。

```
javax.servlet.http.HttpServletRequestWrapper implements ServletRequest
javax.servlet.http.HttpServletResponseWrapper implements ServletResponse
javax.servlet.http.HttpServletRequestWrapper extends
  ServletRequestWrapper implements HttpServletRequest
javax.servlet.http.HttpServletResponseWrapper extends
  ServletResponseWrapper implements HttpServletResponse
```

上述四个类都如同它们实现阶段的接口, 它们的构造函数都必须传入原始的 request 和 response。如果产生的 Wrapper 对象不改变它的内容而直接使用, 它的动作就跟构造函数传入的对象的动作一模一样。

我们该如何利用 Wrapper 呢? 首先我们必须产生一个子类继承 Wrapper, 然后将想修改的方法直接覆写(overriding)。这样一来, 以 Servlet 的观点来看, 就如同原本的 ServletRequest 和

ServletResponse 被继承而产生新的对象一般，这种做法我们称做 Decorator，也就是让外部传入的对象看起来像被继承的技巧。

为了说明 Wrapper 的写法以及运用方式，笔者在这里实际写一个简单的范例。此范例总共有三个文件，即：

RequestWrapperSample: 继承 Wrapper 的类，做修改请求的动作；

WrapperSample: 调用修改请求过程的 filter，将把请求中的部分动作修改掉；

Wrapper.jsp: 测试用的 JSP 文件。

■ *RequestWrapperSample.java*

```
package tw.com.javaworld.CH11;

import javax.servlet.*;
import javax.servlet.http.*;

public class RequestWrapperSample extends HttpServletRequestWrapper {

    public RequestWrapperSample(HttpServletRequest request) {
        super(request);
    }

    public String getMethod() {
        return "POST";
    }

    public String getQueryString() {
        return "www.javaworld.com.tw";
    }
}
```

RequestWrapperSample.java 主要是：当我们调用 `getMethod()` 和 `getQueryString()` 时，回传我们自己设定的参数，而不是照原本 request 传来的参数。原本可能回传 GET 的 `getMethod()` 和 `getQueryString()` 的值分别被改成强制回传 POST 和 `www.javaworld.com.tw`。

■ *WrapperSample.java*

```
package tw.com.javaworld.CH11;

import javax.servlet.*;
import javax.servlet.http.*;

public class WrapperSample implements Filter {

    public void doFilter( ServletRequest request, ServletResponse response,
        FilterChain chain) {
        try {
            RequestWrapperSample wrapper =
                new RequestWrapperSample((HttpServletRequest) request);
            chain.doFilter(wrapper, response);
        } catch (Exception e) { }
    }
}
```

```

    }

    public void init(FilterConfig config) throws ServletException {
    }

    public void destroy() {
    }
}

```

在 doFilter 中我们声明如下:

```

RequestWrapperSample wrapper =
    new RequestWrapperSample( (HttpServletRequest)request );
chain.doFilter(wrapper, response);

```

最后再将 Wrapper 传入 FilterChain 中的 doFilter(), 这样一来, 我们就可以把修改过的请求传给下一个 Filter 或 Servlet、JSP 及其他网页中了。

■ Wrapper.jsp

```

<%@ page contentType="text/html;charset=GB2312" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<html>
<head>
    <title>CH11 - Wrapper.jsp</title>
</head>
<body>

<h2>Wrapper 范例</h2>

${pageContext.request.method}</br>
${pageContext.request.queryString}

</body>
</html>

```

■ web.xml

```

<filter>
    <filter-name>WrapperSample</filter-name>
    <filter-class>tw.com.javaworld.CH11.WrapperSample</filter-class>
</filter>

<filter-mapping>
    <filter-name>WrapperSample</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>

```

图 11-6 为 Wrapper.jsp 执行没有经过 WrapperSample Filter 的结果; 图 11-7 为经过 WrapperSample Filter 的 Wrapper.jsp 执行画面, 可以发现到 getQueryString() 和 getMethod()

回传的参数都被改变了。这节的范例只是简单介绍如何使用 Wrapper，所以可能还看不出来有什么特别的地方，笔者将在之后的章节介绍几个实际运用 Wrapper 的例子。



图 11-6 未经过 Filter 状况下的 Wrapper.jsp

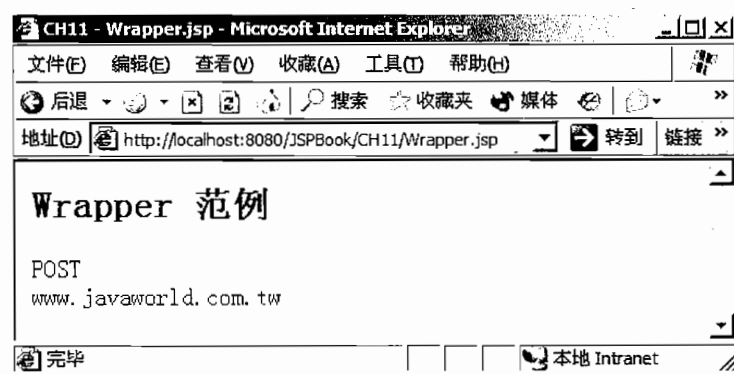


图 11-7 经过 Filter 之后的 Wrapper.jsp

注意

本书范例的 web.xml 中, WrapperSample 的 <filter-mapping> 设定被当做批注, 所以 WrapperSample 不会执行, 即:

```

:
<!--
<filter-mapping>
  <filter-name>WrapperSample</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
-->
:

```

因此若要有上述的结果, 必须将 <!-- 和 --> 删除掉。

接下来, 笔者介绍一个较复杂的范例: 将响应的结果做压缩的动作。

JSP2.0 技术手册

这个范例只有一个程序 `GZIPEncodeFilter.java`。范例中，笔者将响应给用户的内容使用 GZIP 压缩的方式回传给浏览器，而 IE6 和 Netscape 7.1 也都支持 Gzip 的压缩格式。

这个方法在之前就有人提出过，因为是让网页在输出时经过压缩，可以让传输量变小很多，虽然现在的网络频宽对于用来看网页已经绰绰有余，但是文件太大的网页还是会造成一定的影响。经过 Gzip 压缩过的网页，文件大小可以到原本压缩前的 20%。底下这个网页便是经过压缩的例子：http://ftp.boe.tcc.edu.tw/tnc/PHP/php_manual_en.html.gz。读者可以使用下载软件直接下载网页及在 IE 上打开此网页看看。从浏览器中可以看到明确的网页内容，但是直接下载将会发现这是一个压缩文件，压缩后的文件大小便是原本的 20% 左右。接下来我们看主程序：`GZIPEncodeFilter.java`：

■ `GZIPEncodeFilter.java`

```
package tw.com.javaworld.CH11;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.zip.GZIPOutputStream;

public class GZIPEncodeFilter implements Filter {

    public void init(FilterConfig filterConfig) {
    }

    public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain chain) throws IOException, ServletException {
        String transferEncoding =
            getGZIPEncoding((HttpServletRequest) request);
        if (transferEncoding == null) {
            chain.doFilter(request, response);
        } else {
            ((HttpServletResponse) response).setHeader(
                "Content-Encoding", transferEncoding);
            GZIPEncodableResponse wrappedResponse =
                new GZIPEncodableResponse((HttpServletResponse) response);
            chain.doFilter(request, wrappedResponse);
            wrappedResponse.flush();
        }
    }

    public void destroy() {
    }

    private static String getGZIPEncoding(HttpServletRequest request) {
        String acceptEncoding = request.getHeader("Accept-Encoding");
        if (acceptEncoding == null)
            return null;
        acceptEncoding = acceptEncoding.toLowerCase();
        if (acceptEncoding.indexOf("x-gzip") >= 0) {
            return "x-gzip";
        }
    }
}
```

JSP2.0 技术手册


```
        if (acceptEncoding.indexOf("gzip") >= 0) {
            return "gzip";
        }
        return null;
    }

    private class GZIPEncodableResponse extends HttpServletResponseWrapper {
        private GZIPServletStream wrappedOut;
        public GZIPEncodableResponse(HttpServletResponse response)
            throws IOException {
            super(response);
            wrappedOut = new GZIPServletStream(response.getOutputStream());
        }
        public ServletOutputStream getOutputStream() throws IOException {
            return wrappedOut;
        }
        private PrintWriter wrappedWriter;
        public PrintWriter getWriter() throws IOException {
            if (wrappedWriter == null) {
                wrappedWriter = new PrintWriter(
                    new OutputStreamWriter(getOutputStream(),
                        getCharacterEncoding()));
            }
            return wrappedWriter;
        }
        public void flush() throws IOException {
            if (wrappedWriter != null) {
                wrappedWriter.flush();
            }
            wrappedOut.finish();
        }
    }

    private class GZIPServletStream extends ServletOutputStream {
        private GZIPOutputStream outputStream;
        public GZIPServletStream(OutputStream source) throws IOException {
            outputStream = new GZIPOutputStream(source);
        }
        public void finish() throws IOException {
            outputStream.finish();
        }
        public void write(byte[] buf) throws IOException {
            outputStream.write(buf);
        }
        public void write(byte[] buf, int off, int len) throws IOException {
            outputStream.write(buf, off, len);
        }
        public void write(int c) throws IOException {
            outputStream.write(c);
        }
        public void flush() throws IOException {
            outputStream.flush();
        }
    }
}
```

```

        public void close() throws IOException {
            outputStream.close();
        }
    }
}

```

GZIPEncodeFilter.java 程序比之前的程序都要来得复杂, 因此笔者在此分为几个部分来说明此程序, 让读者能了解本程序的制作方法。

```

private static String getGZIPEncoding(HttpServletRequest request) {
    String acceptEncoding = request.getHeader("Accept-Encoding");
    if (acceptEncoding == null)
        return null;
    acceptEncoding = acceptEncoding.toLowerCase();
    if (acceptEncoding.indexOf("x-gzip") >= 0) {
        return "x-gzip";
    }
    if (acceptEncoding.indexOf("gzip") >= 0) {
        return "gzip";
    }
    return null;
}

```

首先笔者在一开始就先检查用户端是不是支持 GZIP 压缩, 可以由 `request.getHeader("Accept-Encoding")` 的方式来得到内容, 只要里面有 `gzip` 字符串或者是 `x-gzip` 字符串, 就表示用户端的浏览方式支持解压缩。

```

private class GZIPEncodableResponse extends HttpServletResponseWrapper {
    private GZIPServletStream wrappedOut;

    public GZIPEncodableResponse(HttpServletResponse response)
        throws IOException {
        super(response);
        wrappedOut = new GZIPServletStream(response.getOutputStream());
    }

    public ServletOutputStream getOutputStream() throws IOException {
        return wrappedOut;
    }

    private PrintWriter wrappedWriter;

    public PrintWriter getWriter() throws IOException {
        if (wrappedWriter == null) {
            wrappedWriter = new PrintWriter(
                new OutputStreamWriter(getOutputStream(),
                    getCharacterEncoding()));
        }
        return wrappedWriter;
    }

    public void flush() throws IOException {
        if (wrappedWriter != null) {
            wrappedWriter.flush();
        }
    }
}

```

JSP2.0 技术手册

```

        wrappedOut.finish();
    }
}

```

另外笔者写了两个类，首先是 `HttpServletResponse` 的 Wrapper 类。在 `HttpServletResponseWrapper` 中，最重要的就是改写原本的输出串流，也就是把原本 `HttpServletResponse` 的 `ServletOutputStream` 和 `PrintWriter` 改写成可以压缩的输出串流。因为任何输出过程一定会先使用这两个方法来得到输出用的对象，因此必须把 `getOutputStream()` 和 `getWriter()` 的回传对象重新取代成压缩用的输出串流才可以。

为什么要覆写 (Overriding) `flush()`？因为 `GZIPOutputStream` 必须在压缩完后调用 `finish()`，因此必须在 `flush()` 中加上调用 `finish()` 的动作。最重要的被改写的输出串流部分，也就是类中的 `wrappedOut` 则是经过改造重新包装过的 `ServletOutputStream`，下面这个类就是继承 `ServletOutputStream` 的类。

```

private class GZIPServletStream extends ServletOutputStream {
    private GZIPOutputStream outputStream;

    public GZIPServletStream(OutputStream source) throws IOException {
        outputStream = new GZIPOutputStream(source);
    }

    public void finish() throws IOException {
        outputStream.finish();
    }

    public void write(byte[] buf) throws IOException {
        outputStream.write(buf);
    }

    public void write(byte[] buf, int off, int len) throws IOException {
        outputStream.write(buf, off, len);
    }

    public void write(int c) throws IOException {
        outputStream.write(c);
    }

    public void flush() throws IOException {
        outputStream.flush();
    }

    public void close() throws IOException {
        outputStream.close();
    }
}

```

新的 `GZIPServletStream` 首先必须继承 `ServletOutputStream`，因为在输出之前，都必须使用 `ServletResponse` 来得到 `ServletOutputStream`。所以在 `GZIPEncodableResponse` 这个类中不可以直接拿 `GZIPOutputStream` 来使用，而是写一个新的输出串流去继承 `ServletOutputStream`，让 `Container` 以为现在使用的就是标准的 `ServletOutputStream`。然后必须去覆写 `finish()`、各种 `write()`、`flush()` 和 `close()`，把它们各自改写成通过 `GZIPOutputStream`

的 `finish()`、`write()`、`flush()` 和 `close()`。这样才可以让这个新的 `ServletOutputStream` 能有压缩的功能（见图 11-8）。

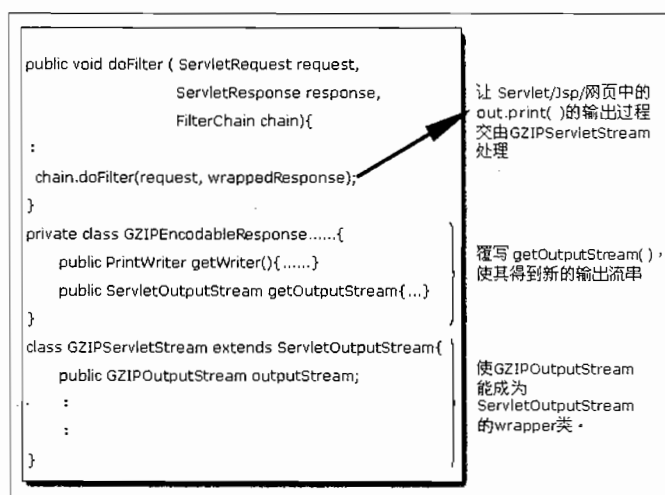


图 11-8 GZIPEncodeFilter 的处理方式

■ web.xml

```

<filter>
    <filter-name>GZIPEncoder</filter-name>
    <filter-class>tw.com.javaworld.CH11.GZIPEncodeFilter</filter-class>
</filter>

<filter-mapping>
    <filter-name>GZIPEncoder</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>

```

`web.xml` 设定为 JSPBook 站台下所有的网页都会经过 `GZIPEncoder Filter`。

注意

本书范例的 `web.xml` 中，`GZIPEncoder` 的 `<filter-mapping>` 设定被当做批注，所以 `GZIPEncoder` 不会执行，即：

```

<!--
<filter-mapping>
    <filter-name>GZIPEncoder</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
-->

```

因此若要有上述的结果，必须将 `<!--` 和 `-->` 删除掉。

在这里笔者任意打开一个网页，可以发现网页仍正常显示出来，但是这样根本看不出有没有经过压缩，读者可以把范例程序中的以下部分修改成为：

```
if (transferEncoding == null) {
    chain.doFilter(request, response);
} else {
    ((HttpServletResponse)response).setHeader(
        "Content-Encoding", transferEncoding);
    GZIPEncodableResponse wrappedResponse =
        new GZIPEncodableResponse((HttpServletResponse)response);

    chain.doFilter(request, wrappedResponse);
    wrappedResponse.flush();
}
```

把上面的程序代码部分修改成任何状况下都要经过压缩，然后经由 telnet 来观看此网页，就会发现全部都是乱码了，也就是说网页已确实经过压缩处理了。

此范例为最典型的做法，我们让任何文件都会经过 GZIP 压缩，但是压缩有时候必须判断哪些内容也就是它的 Content Type 能压缩，而哪些内容不能压缩。例如：image/gif 和 image/jpeg 都是不需要经过压缩的，且某些浏览器的版本更是不支持这些图文件压缩，所以此程序还可以继续改写得更加完善。

11-6 使用 Filter 来解决中文问题

这节笔者将使用 Filter 来解决中文的问题。这里的中文问题是指：Servlet 或 JSP 接收 HTML 窗体的中文数据时所遇到的问题。以往的解决方法：

■ Servlet / JSP

```
request.setCharacterEncoding("GB2312");
request.setCharacterEncoding("GB2312");
```

■ JSTL

```
<fmt:requestEncoding value="GB2312" />
<fmt:requestEncoding value="GB2312" />
```

不过使用传统的方法有一个缺点：即必须对每一个程序都加上上述的程序代码。现在有更好的解决方法，就是使用 Filter 来做时只须写一个 *SetCharacterEncodingFilter* 程序，然后在 *web.xml* 做好路径的设定，这样一来，你就不需要在每一个 JSP 网页中加上解决中文问题的程序代码。

接下来，笔者就要介绍 *SetCharacterEncodingFilter* 程序，因为 Tomcat 已经附上了这个程序的源文件，因此直接拿来使用即可。笔者将节录 *SetCharacterEncodingFilter* 主要程序，先暂时省略一些注释的部分，并且将套件名称改为 *tw.com.javaworld.CH11*：

■ SetCharacterEncodingFilter.java

```
package tw.com.javaworld.CH11;

import java.io.IOException;
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.UnavailableException;

public class SetCharacterEncodingFilter implements Filter {

    protected String encoding = null;
    protected FilterConfig filterConfig = null;
    protected boolean ignore = true;

    public void destroy() {

        this.encoding = null;
        this.filterConfig = null;
    }

    public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain chain) throws IOException, ServletException {

        if (ignore || (request.getCharacterEncoding() == null)) {
            String encoding = selectEncoding(request);
            if (encoding != null)
                request.setCharacterEncoding(encoding);
        }
        chain.doFilter(request, response);
    }

    public void init(FilterConfig filterConfig) throws ServletException {

        this.filterConfig = filterConfig;
        this.encoding = filterConfig.getInitParameter("encoding");
        String value = filterConfig.getInitParameter("ignore");
        if (value == null)
            this.ignore = true;
        else if (value.equalsIgnoreCase("true"))
            this.ignore = true;
        else if (value.equalsIgnoreCase("yes"))
            this.ignore = true;
        else
            this.ignore = false;
    }

    protected String selectEncoding(ServletRequest request) {
```



```

        return (this.encoding);
    }
}

```

`SetCharacterEncodingFilter.java` 提供两个设定值: `encoding` 和 `ignore`。`encoding` 用来设定编码的格式; `ignore` 用来设定是否忽略 client 端所指定的编码, 默认值为 `true`。

■ web.xml

```

:
<filter>
  <filter-name>setCharacterEncoding</filter-name>
  <filter-class>tw.com.javaworld.CH11.SetCharacterEncodingFilter</filter-class>

  <init-param>
    <param-name>encoding</param-name>
    <param-value>GB2312</param-value>
  </init-param>
</filter>

<filter-mapping>
  <filter-name>setCharacterEncoding</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
:

```

`web.xml` 设定为 JSPBook 站台下所有的网页都会经过 `setCharacterEncoding Filter`。

注意

本书范例的 `web.xml` 中, `setCharacterEncoding` 的 `<filter-mapping>` 设定被当做批注, 所以 `setCharacterEncoding` 不会执行, 即:

```

:
<!--
<filter-mapping>
  <filter-name>setCharacterEncoding</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
-->
:

```

因此若要有上述的结果, 必须将 `<!--` 和 `-->` 删除掉。

使用 Filter 解决窗体中文的问题应该是最佳解决方法, 使用简单且方便, 希望对读者有所帮助。

11-7 Listener 接口简介

Web 应用程序员可以利用 Listener 接口, 监听在 Container 中某一个执行程序, 并且根据其应用程序的需求做出适当的响应。

JSP2.0 技术手册

在 Servlet 2.2 和 JSP 1.1 时，新增一个 Listener 接口和一个 Event 类（见表 11-1）。

表 11-1

Listener 接口	Event 类
HttpSessionBindingListener	HttpSessionBindingEvent

在 Servlet 2.3 和 JSP 1.2 时，一口气新增五个 Listener 接口和三个 Event 类（见表 11-2）。

表 11-2

Listener 接口	Event 类
ServletContextListener	ServletContextEvent
ServletContextAttributeListener	ServletContextAttributeEvent
HttpSessionListener	HttpSessionEvent
HttpSessionActivationListener	
HttpSessionAttributeListener	

在 Servlet 2.4 和 JSP 2.0 时，又新增两个 Listener 接口和两个 Event 类（见表 11-3）。

表 11-3

Listener 接口	Event 类
ServletRequestListener	ServletRequestEvent
ServletRequestAttributeListener	ServletRequestAttributeEvent

由上述三个表格可知，目前 Servlet 2.4 和 JSP 2.0 共有八个 Listener 接口和六个 Event 类，其中 HttpSessionAttributeListener 与 HttpSessionBindingListener 皆使用 HttpSessionBindingEvent；HttpSessionListener 和 HttpSessionActivationListener 则都使用 HttpSessionEvent；其余 Listener 对应的 Event 如表 11-4。

表 11-4

Listener 接口	Event 类
ServletContextListener	ServletContextEvent
ServletContextAttributeListener	ServletContextAttributeEvent
HttpSessionListener	HttpSessionEvent
HttpSessionActivationListener	
HttpSessionAttributeListener	HttpSessionBindingEvent
HttpSessionBindingListener	
ServletRequestListener	ServletRequestEvent
ServletRequestAttributeListener	ServletRequestAttributeEvent

上述八个 Listener 接口，笔者将它们分为三类来介绍：

JSP2.0 技术手册

■ 与 ServletContext 有关

- ServletContextListener
- ServletContextAttributeListener

■ 与 HttpSession 有关

- HttpSessionListener
- HttpSessionAttributeListener
- HttpSessionBindingListener
- HttpSessionActivationListener

■ 与 ServletRequest 有关

- ServletRequestListener
- ServletRequestAttributeListener

11-8 ServletContext Listener

ServletContext Listener 接口有两个：

- ServletContextListener
- ServletContextAttributeListener

就单个 Web 站台而言，整个站台的资源都共享一个 `javax.servlet.ServletContext` 类的实体，就如同 JSP 的隐含对象：Application。通过它可以存取应用程序的全局对象以及初始化阶段的变量。

说明

全局对象即为 Application 范围的对象，它的生命周期是从 Container 启动至 Container 关闭才结束。初始阶段的变量是指在 web.xml 中，经由 `<context-param>` 元素所设定的变量，它的范围也是 Application 范围，例如：

web.xml

```
*****
<context-param>
  <param-name>Name</param-name>
  <param-value>browser</param-value>
</context-param>
*****
```

当 Container 启动时，会建立一个 Application 范围的对象，若要在 JSP 网页取得此变量时：

```
String name = (String)application.getInitParameter("Name");
```

或者使用 EL 时:

```
${initParam.name}
```

若是在 Servlet 中, 取得 Name 的值时:

```
String name = (String)ServletContext.getInitParameter("Name");
```

■ javax.servlet.ServletContextListener 接口

一个实现阶段 ServletContextListener 接口的程序, 当 Container 启动时, 程序会自动开始监听的工作, 它首先会调用 contextInitialized() 接收对应的 javax.servlet.ServletContextEvent 事件。ServletContextEvent 对象可以调用 getServletContext() 方法取得 ServletContext 对象。当程序从 Container 移除时, 它又会自动调用 contextDestroyed() 方法, 接收到另一个 ServletContextEvent。表 11-5 为 ServletContextListener 接口的方法。

表 11-5

方 法	说 明
contextInitialized(javax.servlet.ServletContextEvent event)	通知正在收听的对象, 应用程序已经被加载及初始化
contextDestroyed(javax.servlet.ServletContextEvent event)	通知正在收听的对象, 应用程序已经被载出

接下来笔者写个范例, 当 Container 启动时, 实现阶段 ServletContextListener 接口的 MyServletContextListener 类会自动取得 <context-param> 元素设定的初始值并且显示 “Tomcat is running ...”; 当 Container 关闭时, 会显示 “Tomcat is Shutdown ...”。

■ MyServletContextListener

```
package tw.com.javaworld.CH11;

import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;

public class MyServletContextListener implements ServletContextListener {

    public void contextInitialized(ServletContextEvent event) {
        StringInit=
            (String)event.getServletContext().getInitParameter("Init");
        System.out.println("Init: " + Init);

        event.getServletContext().log("Tomcat is running ...");
    }

    public void contextDestroyed(ServletContextEvent event) {
        event.getServletContext().log("Tomcat is Shutdown ...");
    }
}
```

JSP2.0 技术手册

利用 `ServletContext.getInitParameter("Init")` 来取得 `Init` 的值，并且显示出来。

■ web.xml

```
.....
<context-param>
  <param-name>Init</param-name>
  <param-value>browser</param-value>
</context-param>

<listener>
<listener-class>tw.com.javaworld.CH11.MyServletContextListener</listener-
class>
</listener>
.....
```

`MyServletContextListener` 必须在 `web.xml` 中设定，方法如上。当 Container 启动时，会自动调用 `MyServletContextListener` 的 `contextInitialized(event)`；若 Container 关闭时，会自动调用 `contextDestroyed(event)`。下列的结果是当 Container 启动、关闭时，log 文件的内容：

```
2003-11-27 23:22:18 createObjectName with
StandardEngine[Catalina].StandardHost[localhost].StandardContext[/JSPBook]
2003-11-27 23:22:18 preRegister with
Catalina:type=Logger,path=/JSPBook,host=localhost
2003-11-27 23:22:19 Tomcat is running ...
2003-11-27 23:22:55 Tomcat is Shutdown ...
2003-11-27 23:22:55 createObjectName with
StandardEngine[Catalina].StandardHost[localhost].StandardContext[/JSPBook]
```

■ javax.servlet.ServletContextAttributeListener 接口

一个实现阶段 `ServletContextListener` 接口的程序，它能够监听 `Application` 范围的变化。例如：当有对象加入 `Application` 的范围时，实现阶段 `ServletContextListener` 接口的程序会自动被调用通知。除了对象加入 `Application` 范围的情形之外，还有取代、移除的情况，表 11-6 为 `ServletContextAttributeListener` 接口的方法。

表 11-6

方 法	说 明
<code>attributeAdded(javax.servlet.ServletContextAttributeEvent event)</code>	若有对象加入 <code>Application</code> 的范围时，通知正在收听的对象
<code>attributeReplaced(javax.servlet.ServletContextAttributeEvent event)</code>	若在 <code>Application</code> 的范围，有对象取代另一个对象时，通知正在收听的对象
<code>attributeRemoved(javax.servlet.ServletContextAttributeEvent event)</code>	若有对象从 <code>Application</code> 的范围移除时，通知正在收听的对象

ServletContextAttributeEvent 类主要有两个方法:getName()和getValue()。getName()回传新增、移除或取代的属性名称;getValue()则是回传属性的值。因为ServletContextAttributeEvent 类继承于 ServletContextEvent 类,因此它也有 getServletContext()方法。

例如:在 JSP 中,把 Javaworld_url 加入至 Application 范围中:

```
String Javaworld_url = "www.javaworld.com.tw";
application.setAttribute("url", Javaworld);
```

MyServletContextAttributeListener 类的 attributeAdded()会自动被调用:

```
package tw.com.javaworld.CH11;

import javax.servlet.ServletContextAttributeEvent;
import javax.servlet.ServletContextAttributeListener;

public class MyServletContextAttributeListener implements
    ServletContextAttributeListener {

    public void attributeAdded(ServletContextAttributeEvent event) {
        String Attribute_name = event.getName();
        String Attribute_value = (String)event.getValue();
        System.out.println(Name: " + Attribute_name + " Value " + Attribute_value);
    }

    public void attributeReplaced(
        javax.servlet.ServletContextAttributeEvent event) {
        .....
    }

    public void attributeRemoved(
        javax.servlet.ServletContextAttributeEvent event) {
        .....
    }
}
```

实现阶段 ServletContextAttributeListener 接口的类也需要在 web.xml 中设定,如:

■ web.xml

```
.....
<listener>
  <listener-class>
    tw.com.javaworld.CH11.MyServletContextAttributeListener
  </listener-class>
</listener>
.....
```

补充

若在 Application 的范围,有对象取代另一个对象时,此时 getName()和 getValue()是取得取代前的值。

11-9 HttpSession Listener

这节主要介绍与有关 HttpSession 的 Listener 接口，其中包括：

- HttpSessionListener
- HttpSessionAttributeListener
- HttpSessionBindingListener
- HttpSessionActivationListener

■ javax.servlet.http.HttpSessionBindingListener 接口

在第十章中，笔者已经介绍过 HttpSessionBindingListener 和 HttpSessionBindingEvent。当我们的类实现阶段 HttpSessionBindingListener 接口之后，只要对象加入 Session 范围或从 Session 范围中移除时，Container 分别会自动调用下列两个方法：

- valueBound(HttpSessionBindingEvent event)
- valueUnbound(HttpSessionBindingEvent event)

补充

HttpSessionBindingListener 接口是惟一不需要在 web.xml 中设定的 Listener。

■ javax.servlet.http.HttpSessionAttributeListener 接口

HttpSessionAttributeListener 接口和 ServletContextAttributeListener 接口的功用类似，只不过 HttpSessionAttributeListener 接口是监听 Session 范围的变化。HttpSessionAttributeListener 接口一样也有三个方法，如表 11-7。

表 11-7

方 法	说 明
attributeAdded(javax.servlet.http.HttpSessionBindingEvent event)	若有对象加入 Session 的范围时，通知正在收听的对象
attributeReplaced(javax.servlet.http.HttpSessionBindingEvent event)	若在 Session 的范围有对象取代另一个对象时，通知正在收听的对象
attributeRemoved(javax.servlet.http.HttpSessionBindingEvent event)	若有对象从 Session 的范围移除时，通知正在收听的对象

HttpSessionBindingEvent 类主要有三个方法：getName()、getSession()和 getValue()。

JSP2.0 技术手册

getName() 回传新增、移除或取代的属性名称; getSession() 则是回传 HttpSession 对象; getValue() 则是回传属性的值。例如: 在 JSP 中, 把 Javaworld_url 加入至 Session 范围中:

```
String Javaworld_url = "www.javaworld.com.tw";
session.setAttribute("url", Javaworld);
```

MyHttpSessionAttributeListener 类的 attributeAdded() 会自动被调用:

```
package tw.com.javaworld.CH11;

import javax.servlet.http.HttpSession;
import javax.servlet.http.HttpSessionBindingEvent;
import javax.servlet.http.HttpSessionAttributeListener;

public class MyHttpSessionAttributeListener implements
HttpSessionAttributeListener {

    public void attributeAdded(HttpSessionBindingEvent event) {
        String Attribute_name = event.getName( );
        String Attribute_value = (String)event.getValue( );
        HttpSession session = (HttpSession)event.getSession( );
        System.out.println(Name: " + Attribute_name + " Value " + Attribute_value);
    }

    public void attributeReplaced(
        javax.servlet.http.HttpSessionBindingEvent event) {
        .....
    }

    public void attributeRemoved(
        javax.servlet.http.HttpSessionBindingEvent event) {
        .....
    }
}
```

同样地, MyHttpSessionAttributeListener 必须也要在 web.xml 中设定:

■ web.xml

```
.....
<listener>
  <listener-class>
    tw.com.javaworld.CH11.MyHttpSessionAttributeListener
  </listener-class>
</listener>
.....
```

HttpSessionAttributeListener 和 HttpSessionBindingListener 两者的功用都蛮类似, 不过两者在某些地方上, 有些许的不同:

- HttpSessionAttributeListener 需要在 web.xml 中设定; 而 HttpSessionBindingListener 不需要。
- HttpSessionAttributeListener 监听 Web 站台所有 Session 范围的变化; 而

JSP2.0 技术手册

HttpSessionBindingListener 只单纯监听实现阶段它的对象。

■ javax.servlet.http.HttpSessionListener 接口

HttpSessionListener 接口和 ServletContextListener 接口类似, 当有 Session 产生或是消灭, 会自动调用 sessionCreated() 和 sessionDestroyed(), 如表 11-8。

表 11-8

方 法	说 明
sessionCreated(javax.servlet.http.HttpSessionEvent event)	通知正在收听的对象, Session 已经被加载及初始化
sessionDestroyed(javax.servlet.http.HttpSessionEvent event)	通知正在收听的对象, Session 已经被载出

HttpSessionEvent 类主要的方法: getSession()。可以使用 getSession() 回传一个 Session 对象。

```
package tw.com.javaworld.CH11;

import javax.servlet.http.HttpSession;
import javax.servlet.http.HttpSessionEvent;
import javax.servlet.http.HttpSessionListener;

public class MyHttpSessionListener implements HttpSessionListener {

    private static int session_count = 0;

    public void sessionCreated(HttpSessionEvent event) {
        HttpSession session = (HttpSession)event.getSession();
        .....
        session_count ++;
    }

    public void sessionDestroyed(javax.servlet.http.HttpSessionEvent event) {
        .....
        session_count --;
    }
}
```

同样地, MyHttpSessionListener 也必须要在 web.xml 中设定:

■ web.xml

```
.....
<listener>
.....
<listener-class>tw.com.javaworld.CH11.MyHttpSessionListener</listener-
class>
```

JSP2.0 技术手册

```
</listener>
.....
```

■ javax.servlet.http.HttpSessionActivationListener 接口

HttpSessionActivationListener 接口主要用于：同一个 Session 转移至不同 JVM 的情形，例如：负载均衡(Load Balancing)机制，这些 JVM 可以在同一台机器或是分散在网络中的多台机器。

当 Session 被储存起来，并且等待转移至另一个 JVM，这段时间称为失效状态(Passivate)，若 Session 中的属性对象实现阶段 HttpSessionActivationListener 接口时，Container 会自动调用 sessionWillPassivate()方法，通知该对象的 Session 已变成失效状态。当 Session 被转移至其他 JVM 之后，它又成为有效状态(Activate)，此时 Container 会自动调用 sessionDidActivate()方法，通知该对象的 Session 已变成有效状态。HttpSessionActivationListener 接口的两个方法如表 11-9。

表 11-9

方 法	说 明
sessionDidActivate(javax.servlet.http.HttpSessionEvent event)	通知正在收听的对象，它的 Session 已经变为有效状态
sessionWillPassivate(javax.servlet.http.HttpSessionEvent event)	通知正在收听的对象，它的 Session 已经变为无效状态

HttpSessionEvent 类主要的方法 getSession()。可以使用 getSession()回传一个 Session 对象。

```
package tw.com.javaworld.CH11;

import javax.servlet.http.HttpSession;
import javax.servlet.http.HttpSessionEvent;
import javax.servlet.http.HttpSessionActivationListener;

public class MyHttpSessionActivationListener implements
HttpSessionActivationListener{

    public void sessionDidActivate(javax.servlet.http.HttpSessionEvent event){
        .....
    }

    public void sessionWillPassivate(HttpSessionEvent event) {
        HttpSession session = (HttpSession)event.getSession();
    }
}
```

同样地，MyHttpSessionActivationListener 必须也要在 web.xml 中设定：

■ web.xml

```
.....
<listener>
  <listener-class>
    tw.com.javaworld.CH11.MyHttpSessionActivationListener
  </listener-class>
</listener>
```

JSP2.0 技术手册

```
</listener>
.....
```

11-10 ServletRequest Listener

ServletRequest Listener 是 Servlet 2.4 新增的功能，它主要有两个接口：

- ServletRequestListener
- ServletRequestAttributeListener

■ javax.servlet.ServletRequestListener 接口

ServletRequestListener 接口和 ServletContextListener 接口类似，当有请求产生或是消灭，会自动调用 requestInitialized() 和 requestDestroyed()，如表 11-10。

表 11-10

方 法	说 明
requestInitialized(javax.servlet.ServletRequestEvent event)	通知正在收听的对象，ServletRequest 已经被加载及初始化
requestDestroyed(javax.servlet.ServletRequestEvent event)	通知正在收听的对象，ServletRequest 已经被载出

ServletRequestEvent 类主要的方法有两个：getServletContext() 和 getServletRequest()。getServletContext() 回传一个 ServletContext 对象；而 getServletRequest() 回传一个 ServletRequest 对象。

```
package tw.com.javaworld.CH11;
```

```
import javax.servlet.ServletContext;
import javax.servlet.ServletRequest;
import javax.servlet.ServletRequestEvent;
import javax.servlet.ServletRequestListener;
```

```
public class MyServletRequestListener implements ServletRequestListener {

    public void requestInitialized(ServletRequestEvent event) {
        ServletContext application = (ServletContext)event.getServletContext();
        ServletRequest request = (ServletRequest)event.getServletRequest();
        .....
    }

    public void requestDestroyed(ServletRequestEvent event) {
        .....
    }
}
```

JSP2.0 技术手册

同样地，MyServletRequestListener 必须也要在 `web.xml` 中设定：

■ `web.xml`

```
.....
<listener>
  <listener-class>tw.com.javaworld.CH11.MyServletRequestListener
</listener-class>
</listener>
.....
```

■ `javax.servlet.ServletRequestAttributeListener` 接口

`ServletRequestAttributeListener` 接口和 `ServletContextAttributeListener` 接口的功用类似，只不过 `ServletRequestAttributeListener` 接口是监听 `Request` 范围的变化。`ServletRequestAttributeListener` 接口一样也有三个方法，如表 11-11。

表 11-11

方 法	说 明
<code>attributeAdded(javax.servlet.ServletRequestAttributeEvent event)</code>	若有对象加入 <code>Request</code> 的范围时，通知正在收听的对象
<code>attributeReplaced(javax.servlet.ServletRequestAttributeEvent event)</code>	若在 <code>Request</code> 的范围有对象取代另一个对象时，通知正在收听的对象
<code>attributeRemoved(javax.servlet.ServletRequestAttributeEvent event)</code>	若有对象从 <code>Request</code> 的范围移除时，通知正在收听的对象

`ServletRequestAttributeEvent` 类主要有两个方法：`getName()`和`getValue()`。`getName()`回传新增、移除或取代的属性名称；`getValue()`则是回传属性的值。因为 `ServletRequestAttributeEvent` 类继承于 `ServletRequestEvent` 类，因此它也有 `getServletContext()`和`getRequest()`。

例如：在 JSP 中，把 `Javaworld_url` 加入至 `Request` 范围中：

```
String Javaworld_url = "www.javaworld.com.tw";
request.setAttribute("url", Javaworld);
```

`MyServletRequestAttributeListener` 类的 `attributeAdded()`会自动被调用：

```
package tw.com.javaworld.CH11;

import javax.servlet.ServletContext;
import javax.servlet.ServletRequest;
import javax.servlet.ServletRequestAttributeEvent;
import javax.servlet.ServletRequestAttributeListener;

public class MyServletRequestAttributeListener implements
ServletRequestAttributeListener {
```



```
public void attributeAdded(ServletRequestAttributeEvent event) {
    String Attribute_name = event.getName( );
    String Attribute_value = (String)event.getValue( );
    ServletContext application = (ServletContext)event.getServletContext( );
    ServletRequest request = (ServletRequest)event.getServletRequest( );
    System.out.println(Name: " + Attribute_name + " Value " + Attribute_value);
}

public void attributeReplaced(ServletRequestAttributeEvent event) {
    .....
}

public void attributeRemoved(ServletRequestAttributeEvent event) {
    .....
}
}
```

同样地，MyServletRequestAttributeListener 必须也要在 *web.xml* 中设定：

■ *web.xml*

```
.....
<listener>
  <listener-class>
    tw.com.javaworld.CH11.MyServletRequestAttributeListener
  </listener-class>
</listener>
.....
```


12

第十二章

JSP 执行环境与开发工具

本章主要分为两大主题讨论：JSP 执行环境和开发工具。

JSP 执行环境即为 JSP Container。笔者在撰写本书时，市面上只有 Tomcat 5.0.16 和 Resin 3.04 可以执行 Servlet 2.4 和 JSP 2.0，但是 Tomcat 5.0 是免费的且开放源代码，因此笔者将选用 Tomcat 5.0.16。

本章 JSP 开发工具的介绍主要分为阳春型和 IDE 工具型，如：编辑器搭配 ant，而 IDE 工具型主要介绍 Eclipse。本章分 6 节为各位读者介绍：

- 12-1 Tomcat 5.0 的介绍
- 12-2 JSP 开发工具介绍
- 12-3 Eclipse 简介与安装
- 12-4 使用 Eclipse 开发 Hello World
- 12-5 使用 Eclipse 开发 Web Application
- 12-6 使用 Eclipse 开发 Web Application(2)

JSP2.0 技术手册

12-1 Tomcat 5.0 的介绍

Tomcat 的目前版本为 5.0.16, 它是由 JavaSoft 和 Apache 开发团队共同提出合作计划(Apache Jakarta Project)下的产品。Tomcat 支持 Servlet 2.4 和 JSP 2.0 并且免费使用。除此之外, Tomcat 还开放源代码供大众下载使用, 有兴趣的读者可以至:

<http://jakarta.apache.org/site/sourceindex.cgi>

补充
第二次美伊战争时, 笔者观看 CNN 的战况报导, 突然发现 F-14 Tomcat 的字眼, 报章杂志中译为 F-14 雄猫战斗机。雄猫的确比汤姆猫威武许多, 假若报章杂志也将 F-14 Tomcat 译为 F-14 汤姆猫战斗机时, 那……

12-1-1 Tomcat 5 目录结构

首先, 我们从 Tomcat 的目录结构谈起, 如表 12-1 所示。

表 12-1

目 录	说 明
\bin	主要为启动 (<i>startup.bat</i> , <i>startup.sh</i>) 关闭 (<i>shutdown.bat</i> , <i>shutdown.sh</i>) 等 .bat、.sh 文件
\common	放置在此目录的 JAR、类文件, 能让所有 \webapps 下的站台和 Tomcat 内部程序所使用, 例如: \common\lib 是放置 JDBC Driver 最佳的地方 (JDBCRealm、JDBC DataSource)
\conf	Tomcat 主要的设定文件, 其中包括 <i>server.xml</i> 、 <i>web.xml</i> 、 <i>tomcat-user.xml</i> 和 <i>catalina.policy</i>
\logs	放置 Tomcat 日志文件的目录, 其中包括 <i>catalina_log</i> 、 <i>localhost_admin_log</i> 、 <i>localhost_log</i> 、 <i>stderr.log</i> 和 <i>stdout.log</i>
\server	放置 Tomcat 后端管理接口的站台, 包括 admin 和 manager
\shared	放置在此目录的 JAR、类文件, 能让所有 \webapps 下的站台所使用, 但是无法让 Tomcat 内部程序使用
\src	放置 Tomcat 相关的源代码, 其中包括 jakarta-servletapi-5、jakarta-tomcat-5、jakarta-tomcat-catalina、jakarta-tomcat-connectors 和 jakarta-tomcat-jasper
\temp	Tomcat 暂存区目录
\webapps	放置 web 站台的目录, 默认有 <i>ROOT</i> 、 <i>jsp-examples</i> 、 <i>servlets-examples</i> 和 <i>tomcat-docs</i>
\work	放置 JSP 网页转译成 Servlet 类的目录

图 12-1 为 Tomcat 5.0 的目录结构:

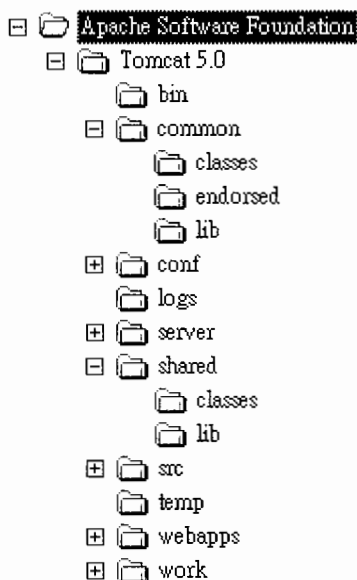


图 12-1 Tomcat 5.0 目录结构

12-1-2 建立一个新的站台，部署 Servlet/JSP/JavaBean

了解 Tomcat 5.0 的目录结构之后，下面就建立一个 Test 的站台来作为范例：

■ 修改 server.xml

一开始修改 `{Tomcat_Install}\conf\server.xml` 将下列这段程序代码加在 `server.xml` 尾端的 `<!-- Tomcat Root Context -->` 之后，然后再存盘，重新启动 Tomcat。

■ server.xml

```
<Server>
.....
  <Service>
    .....
    <Engine>
      .....
      <Host>
        .....
        <!-- Tomcat Root Context -->
        <Context path="/Test" docBase="Test"
          debug="0" crosscontext="true" reloadable="true">
        </Context>
      </Host>
    </Engine>
  </Service>
</Server>
```

JSP2.0 技术手册

我们要建立一个站台,其中 `path="/Test"` 的意思是说:将新建一个站台,名称为 `Test`,则它的 URL 位置就是 `http://server IP or Domain /Test/`。`docBase="Test"` 的意思是: `Test` 站台本地端的目录放置在 `{Tomcat_Install}\webapps\Test`。`crossContext="true"`:表示允许你通过 `ServletContext.getContext()` 存取其他的站台。`debug` 则是设定 `debug level`, `0` 表示提供最少的信息, `9` 表示提供最多的信息。`reloadable` 则表示 `Tomcat` 在执行时,修改过的 `class` 或 `web.xml` 都会自动重新加载,不需再重新执行 `Tomcat`。

存盘重新启动 `Tomcat` 后,我们在 `{Tomcat_Install}\webapps\` 下手动建立一个 `Test` 的目录,而 JSP 程序就放在此目录下。然后在 `Test` 目录下再产生一个 `WEB-INF` 目录,并在其下再建立 `classes`、`lib` 目录和 `web.xml`,其中 `classes`、`lib` 目录都是用来放置 `Servlet`、`JavaBean` 或 `API`。因此,最后的目录结构为:

```
{Tomcat_Install}\webapps\Test
{Tomcat_Install}\webapps\Test\WEB-INF\
{Tomcat_Install}\webapps\Test\WEB-INF\lib
{Tomcat_Install}\webapps\Test\WEB-INF\classes
{Tomcat_Install}\webapps\Test\WEB-INF\web.xml
```

■ 修改 web.xml

目前 `Test` 站台并无任何的 `Servlet`、`JSP`、`Filter` 或 `Tag`,所以先暂时不需要太多的设定,只要先有下列的声明即可:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
  version="2.4">

  <display-name>Test</display-name>
  <description>Test Web Application</description>

  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>
</web-app>
```

上述的 `web.xml` 中,除了一般的声明之外,笔者还加上 `<welcome-file-list>` 来设定 `Test` 站台的默认首页。当执行 `http://localhost:8080/Test/` 时,它会自动找寻 `index.jsp`、`index.html` 作为首页,若找不到,则会产生“HTTP 404 找不到网页”的错误。

■ 在 Test 下,执行 Servlet

假设你写好一个套件名称为 `tw.com.javaworld` 的 `Hello Servlet` 程序时,你要将 `Servlet` 程序

JSP2.0 技术手册

放置到:

```
{Tomcat_Install}\webapps\Test\WEB-INF\classes\tw\com\javaworld\Hello.class
```

的目录下后, 并且在 *web.xml* 增加下列的设定:

```
<servlet>
  <servlet-name>Hello</servlet-name>
  <servlet-class>tw.com.javaworld.Hello</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>Hello</servlet-name>
  <url-pattern>/Hello</url-pattern>
</servlet-mapping>
```

然后开启浏览器输入:

```
http://localhost:8080/Test/Hello
```

就可以顺利执行 Hello Servlet 程序。

■ 在 Test 下, 执行 JSP

将 *Hello.jsp* 程序放置在:

```
{Install_Tomcat}\Tomcat 5.0\webapps\Test\Hello.jsp
```

你只要开启浏览器输入:

```
http://localhost:8080/Test/Hello.jsp
```

就可以顺利执行你的 JSP 程序了。

■ 在 Test 下, 执行 JSP + JavaBean

JSP 放置的路径和上述相同; 而 JavaBean 的放置路径就如同 Servlet 一样, 形成:

```
{Install_Tomcat}\Tomcat
5.0\webapps\Test\WEB-INF\classes\date\JspCalendar.class
```

因此你在 JSP 调用 JavaBean 的方法如下:

```
<jsp:useBean id="clock" scope="page" class="dates.JspCalendar" />
```

dates 就是套件的名称, JspCalendar 就是 JavaBean 的 class 名称。JavaBean 的使用在第八章有更详细的说明。

12-2 JSP 开发工具介绍

古人说：“工欲善其事，必先利其器”。因此在开发 JSP 时，如果有好的开发环境，开发的效率和品质也会有一定的水准。笔者在此介绍目前现有可以用来开发 JSP 的工具，并且大略说明如何使用这些开发工具来快速开发 JSP 的程序。

■ 阳春型

一般来说，最阳春的开发工具就是一些编辑软件，如笔记簿、UltraEdit 等等。你只要把程序写完之后，再把扩展名取为 `.jsp` 之后就完成了，但是它并不能帮你做 Debug 的工作，只能先把程序放到执行环境中来测试你所写的程序是否正确。不过根据笔者的经验，千万不要以为执行环境会帮你做 Debug 的工作，有时它只会显示错误的信息，例如：大家以后碰到最常见的 500 Internal Server Error，但是并不会告诉你到底是哪一行程序有问题，因此当初笔者在开发时，曾经因为手误打错字，白白浪费时间。

■ IDE 工具型

目前有许多的 Java 开发工具，其中多数人使用且免费和开放源代码的包含：Sun 支持的 NetBean(<http://www.netbeans.org/index.html>)、IBM 支持的 Eclipse (<http://www.eclipse.org>)。两者各有优势，不过目前使用 Eclipse 的人较多。

另外其他较知名的开发工具有 Borland JBuilderX、Oracle JDeveloper 10g、Sun Java Studio、BEA WorkShop、IBM Websphere Studio 等等。

本书接下来将介绍 Eclipse 的使用。

12-3 Eclipse 简介与安装

Eclipse 是在 Eclipse Project (<http://www.eclipse.org>)所公开的程序开发工具，由 1999 年 4 月的 OTI(Object Technology International)和 IBM 的共同设计所诞生。2001 年 10 月 Eclipse Ver 1.0 发表，到了 2002 年 6 月 Ver 2.0 被发表了。而笔者撰写本章节的时候则是 Ver 2.1.2。

Eclipse 的特征在于可以使用 Plug-in 来扩展它的功能。图 12-2 中，Eclipse 以 JVM 为底，包含了 Platform, JDT(Java Development Tools), PDE(Plug-in Development Environment)这几层。而利用 PDE 来开发扩展 Eclipse 功能的 Plug-in 也是 Eclipse 的一大特征(<http://www.eclipse-plugins.2y.net/eclipse/index.jsp>)。现在 Eclipse 已经有三百个以上的 Plug-in 可以使用。由于以上的优势再加上 Eclipse 是开放源代码的关系，让 Eclipse 以迅雷不及掩耳之速席卷了所有 Java Programmer 的桌面。

JSP2.0 技术手册

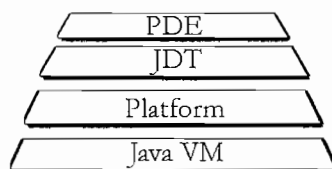


图 12-2 Eclipse 架构

除了上述让 Eclipse 迅速普及的特点之外, XP(eXtreme Programming)的出现也让 Eclipse 犹如虎添翼。XP 的特点在于软件开发上特别重视通过工具来迅速地开发。为了达到 XP 所要达到的目标, 需要实践许多规范, 如果用了如 Eclipse 般的 IDE, 就更能有效地达到 XP 要求的成果。尤其 Eclipse 中一开始就支持了 JUnit, 让我们可以迅速地达到 Test-First(测试优先)的开发目标。

Eclipse 可以从本书附赠的光盘中取得, 版本为 2.1.2, 当然也可以直接从网络上下载安装。

■ Eclipse 的下载

Eclipse 的最新版本可以由 *eclipse.org* 的网站(<http://www.eclipse.org>, 如图 12-3)下载。

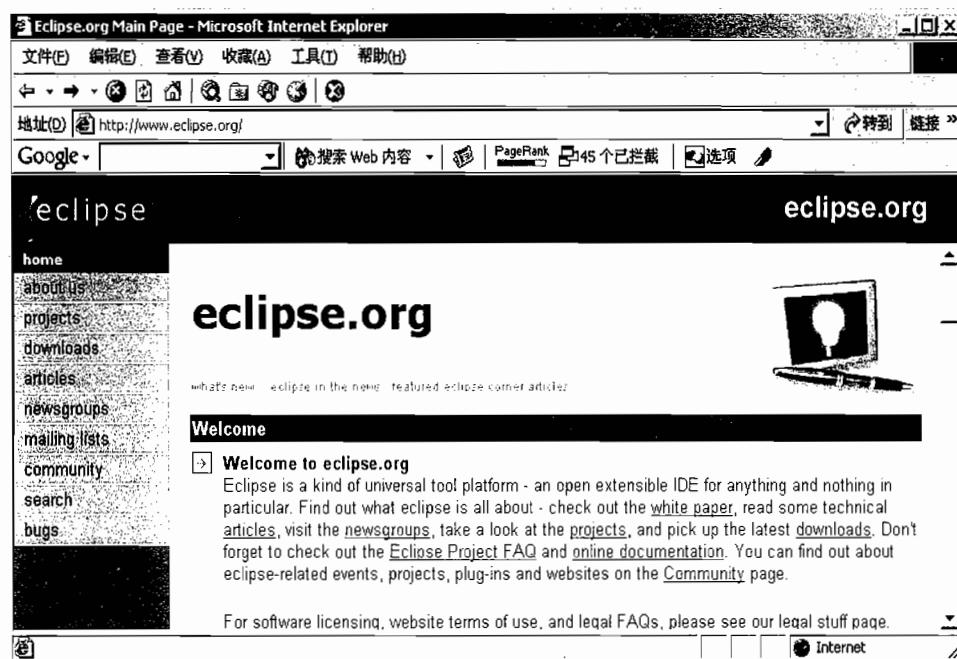


图 12-3 Eclipse 官方首页

选取左边的【downloads】到 Eclipse 的下载页面(<http://www.eclipse.org/downloads/>)。在此页中

会列出提供下载的网址。选择好最近的下载点后, 就会到达现在公开的最新版本的下载页面, 如图 12-4。

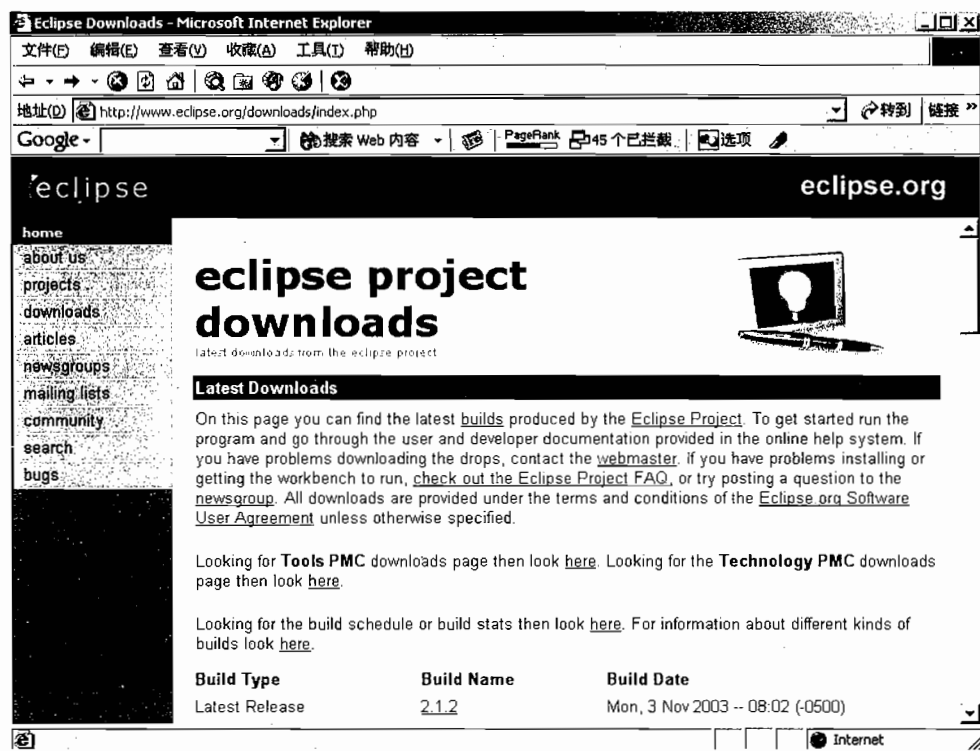


图 12-4 Eclipse 下载位置

在这里选择下载 Latest Release 的 Ver2.1.2, 到达下个页面后依据您的操作系统选择 Eclipse SDK 下载。

■ J2SDK 的下载

到 SUN 的网站(<http://java.sun.com>)下载 J2SDE1.4.2 的 JDK。

■ Eclipse 的安装

笔者安装的环境与版本如下:

OS: Microsoft Windows XP

JDK: *j2sdk-1_4_2_03-windows-i586-p.exe*

Eclipse: *eclipse-SDK-2.1.2-win32.zip*

JSP2.0 技术手册

首先安装完 JDK，接下来就是启动 Eclipse 了。将 `eclipse-SDK-2.1.2-win32.zip` 解压缩，在这里笔者将 eclipse 解压缩放在 C:\下，打开命令行，执行

```
C:\eclipse\eclipse.exe -data C:\workspace
```

执行后将会在 C:\下产生一个文件夹 `workspace`。并且在画面上出现 Eclipse 的画面，如图 12-5。

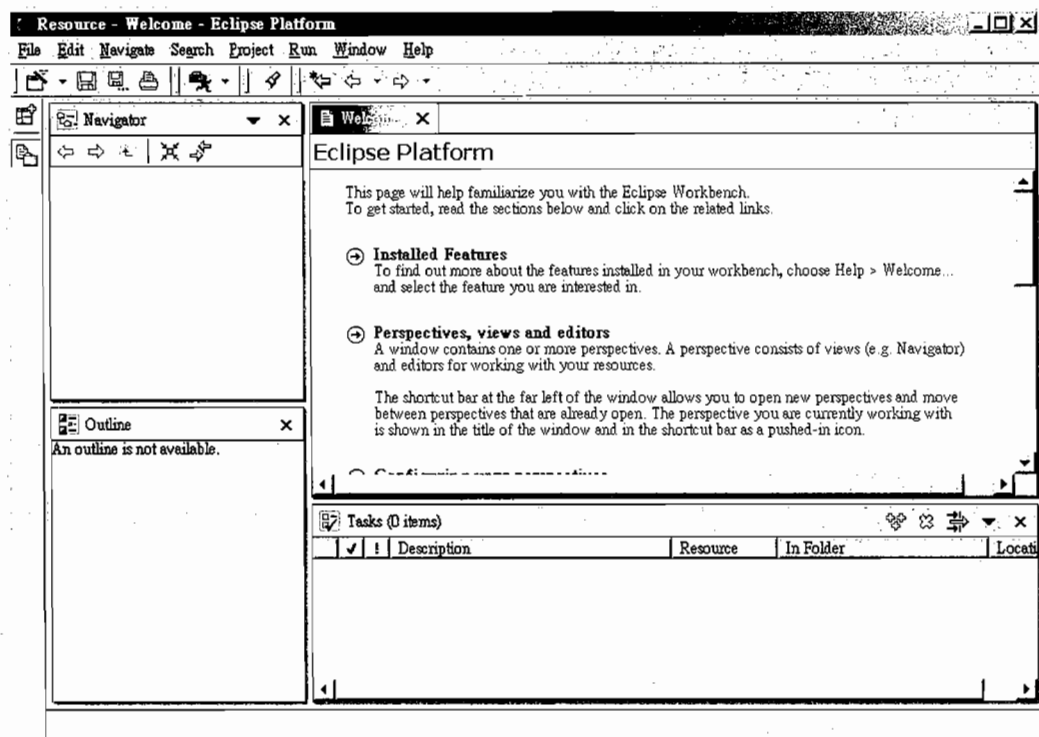


图 12-5 Eclipse 启动后的画面

注意

如果省略了 `-data`，将会在 eclipse 文件夹下自动产生 `workspace` 文件夹。

12-4 使用 Eclipse 开发 Hello World

在开发 JAVA 程序之前，选择【Window | Open Perspective | Java】切换到 Java perspective。此步骤会将 Navigator 变成 Package Explore，而在 Editor 右边会多出个 Outline，工具栏也会变成 Java 程序开发用的工具栏，如图 12-6。

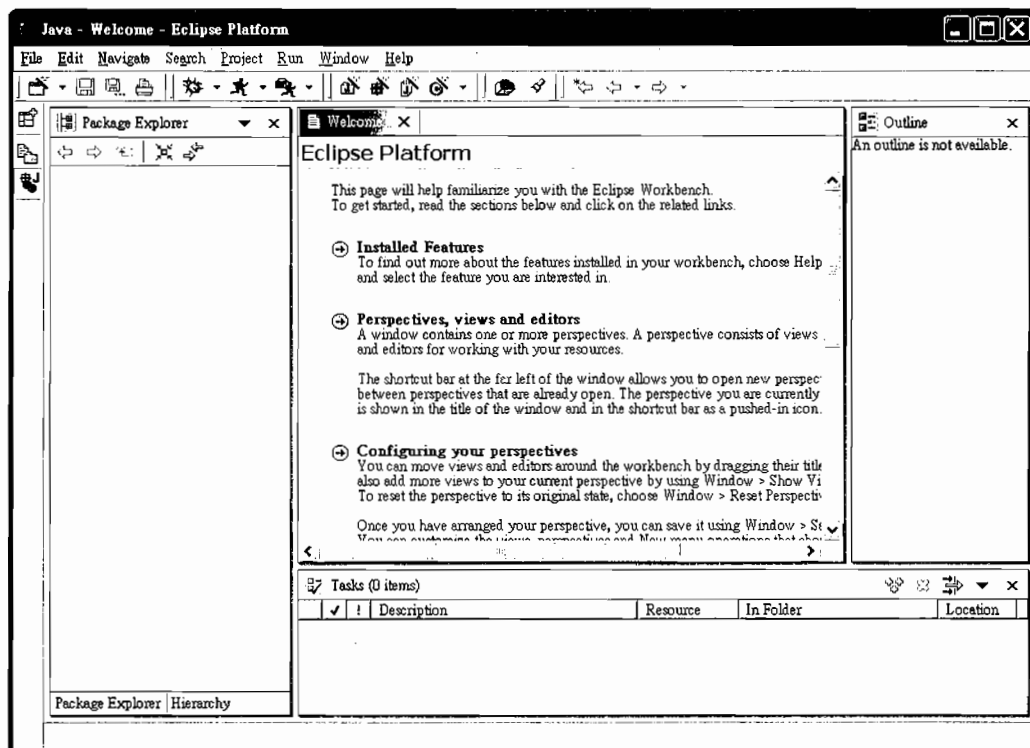


图 12-6 切换到 Java perspective

■ Java Project 的产生

在 Eclipse 中，所有文件都必须制作于 project 的文件夹下。Java Project 由 New Java Project 向导所产生，先选择【File | New | Project】，然后选择【Java Project】，如图 12-7。

选择【Java Project】后点击【Next】，接下来在 Project name 的地方填上 JSPBook，点击【Finish】按钮后产生 JSPBook 的 project。

产生完 project 后选择【File | New | Package】，确定好 Source Folder 字段为 JSPBook 后在 Name 的字段填上 tw.com.javaworld.CH12，按下【Finish】。接下来在 Package Explore 中会新增一个 tw.com.javaworld.CH12 的套件。因为此套件中尚无任何东西，所以会发现 icon 的颜色为白色；当套件中有文件以后 icon 将自动变为橙色。

接下来撰写 Eclipse 上的第一个 Java 程序。第一个 Java 程序为 *HelloWorld.java*。在 Package Explore 中选择 tw.com.javaworld.CH12 的套件的状态下，点击【File | New | Class】后产生如图 12-8 的画面。

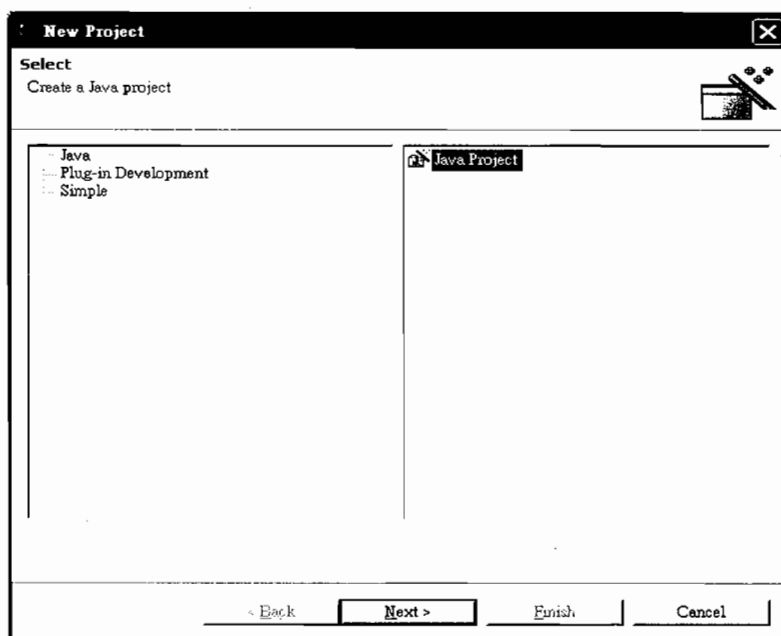


图 12-7 New Project 画面

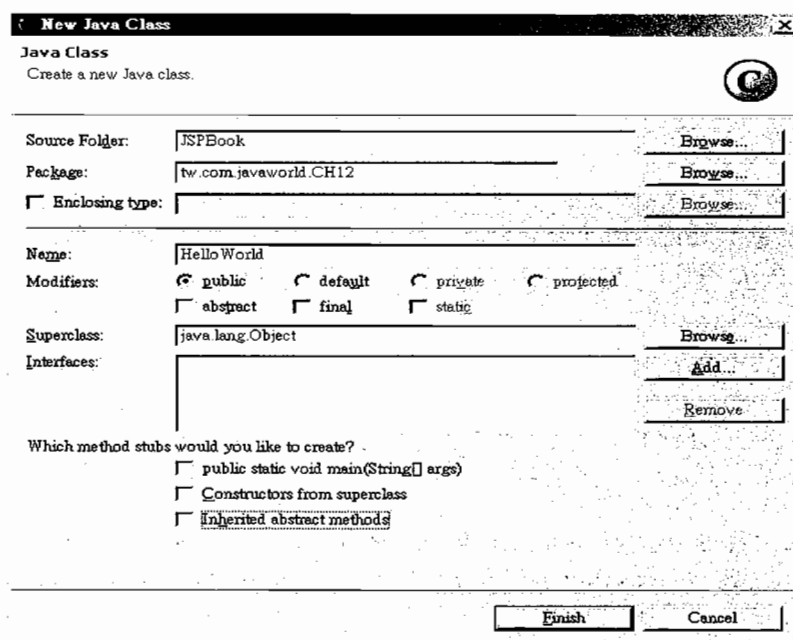


图 12-8 New Java Class

你会发现 Source Folder 和 Package 都已填上 JSPBook 和 tw.com.javaworld.CH12 了，确认后在 Name 中输入 HelloWorld，然后最下面的 Which method stubs would you like to create 的 check box 都不选取，最后按下【Finish】。产生如图 12-9 的结果：

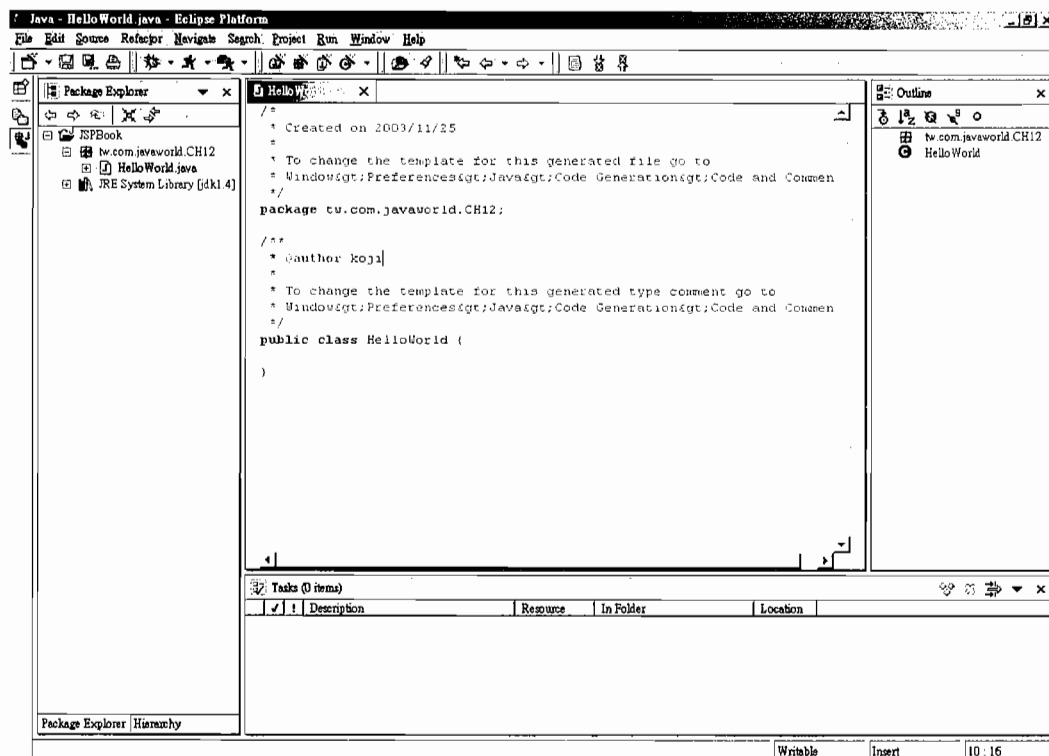


图 12-9 产生 HelloWorld 类

画面的中间可以编辑 *HelloWorld.java*，我们将写一个在 console 下显示“Hello World!!”的程序。程序代码内容如下：

■ HelloWorld.java

```


/*
 * Created on 2003/11/25
 *
 * To change the template for this generated file go to
 * Window>Preferences>Java>Code Generation>Code and Comments
 */
package tw.com.javaworld.CH12;

/**
 * @author morchory
 *
 * To change the template for this generated type comment go to
 * Window>Preferences>Java>Code Generation>Code and Comments
 */

```

JSP2.0 技术手册

```
*/  
public class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello World!!");  
    }  
}
```

最后在工具栏上，点击【执行】按钮  右边的下拉符号，选择【Run As | Java Application】，如图 12-10。另外，也可以从菜单栏选择【Run | Run As | Java Application】。

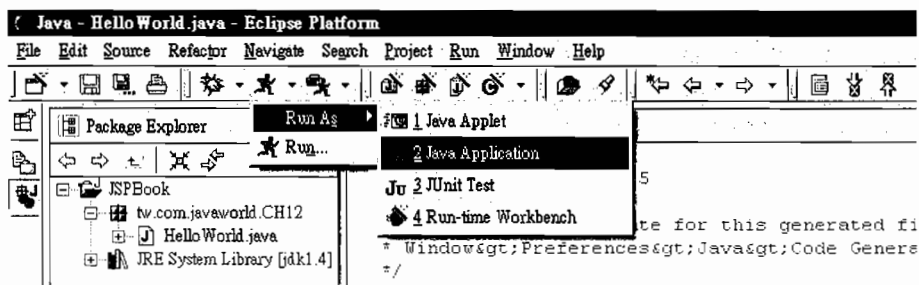


图 12-10 Run As Java Application

按下后会在 editor 下面转换成 Console 并且显示“Hello World!!”，如图 12-11：

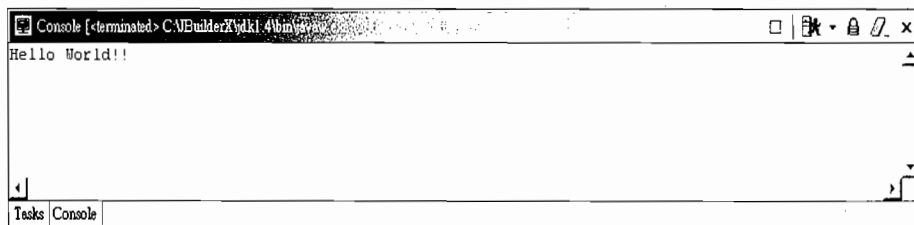


图 12-11 HelloWorld 的执行结果

12-5 使用 Eclipse 开发 Web Application

接下来要介绍的是如何使用 Eclipse 来开发 Web Application。因为 Eclipse 本身并无提供开发 Web Application 的工具，所以我们必须去另外抓取 plug-in 来让 Eclipse 拥有这些功能。因此在一开始，笔者先介绍必须安装哪些软件，才能让我们使用 Eclipse 来开发 Web Application。要使用的 plug-in 如下：

- Sysdeo Eclipse Tomcat Launcher Ver 2.2.1
- SolarEclipse Suite of Development Tools ver 0.4.0

另外笔者使用的环境与版本如下：

JSP2.0 技术手册

- Microsoft Windows XP
- JDK 1.4.2
- Eclipse 2.1.2
- Tomcat 5.0.16

■ Sysdeo Eclipse Tomcat Launcher 的安装

Sysdeo Eclipse Tomcat Launcher 是用来让我们从 Eclipse 启动 Tomcat 且直接在 Tomcat 上撰写程序的 plug-in。此 plug-in 可以从网络上或本书所附的光盘中取得。此 plug-in 所在的网址为 <http://www.sysdeo.com/eclipse/tomcatPlugin.html>。在这里笔者使用的为 tomcatPluginV221.zip 的版本，下载后解压缩并放到 Eclipse 安装目录下的 *plugins* 文件夹中。以上步骤就将 Sysdeo Eclipse Tomcat Launcher 安装完毕了。

■ Tomcat Project 的设定

开启菜单上的【window | Customize Perspective】，打开 Customize Perspective 的画面，点击【others】分类可以看到 Tomcat 的选项，将它打勾，如图 12-12：

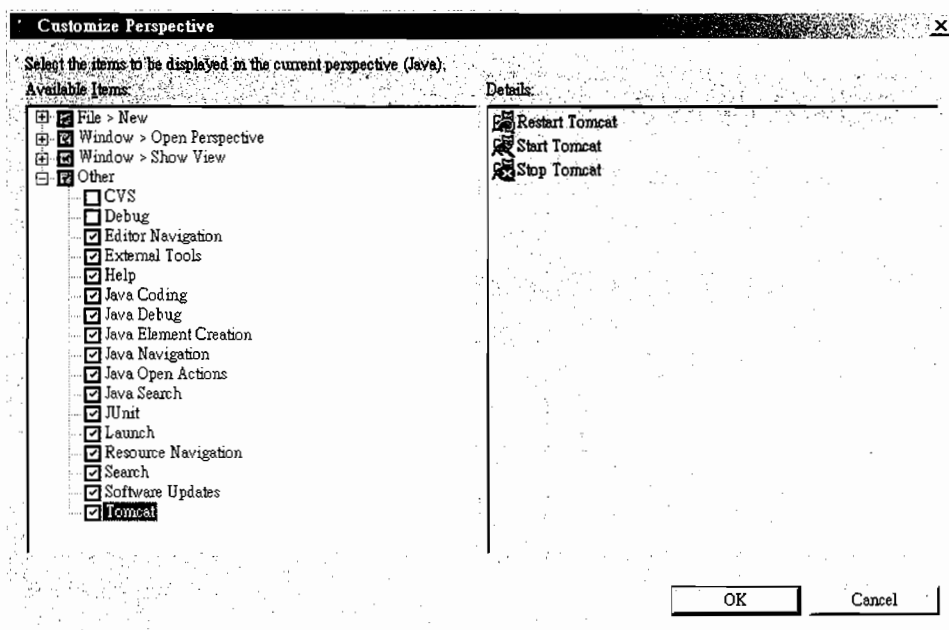


图 12-12 Customize Perspective

另外从【File | New】分类可以看到【Tomcat Project】这个选项，也将其打勾，如图 12-13：

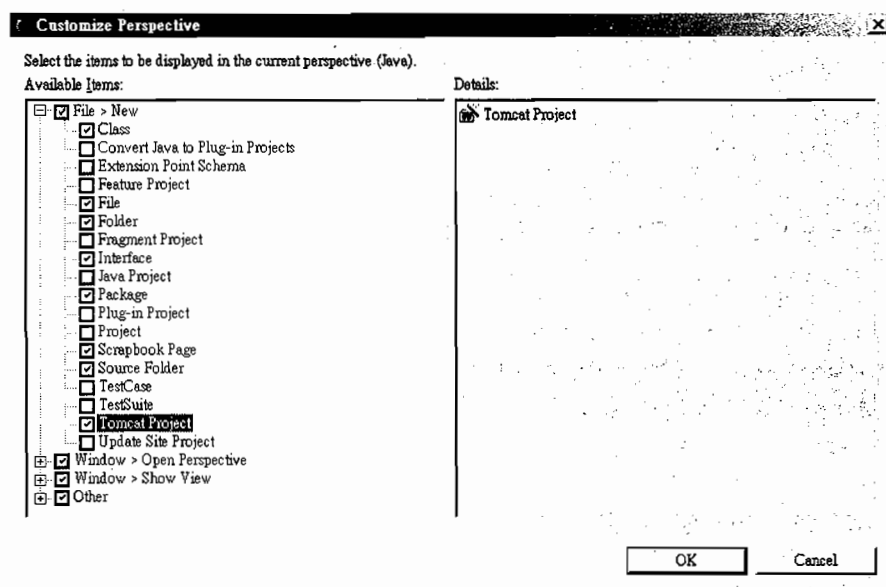


图 12-13 勾选【Tomcat Project】

照以上设定以后，将可以直接从菜单的【File | New】点击【Tomcat Project】，如图 12-14：

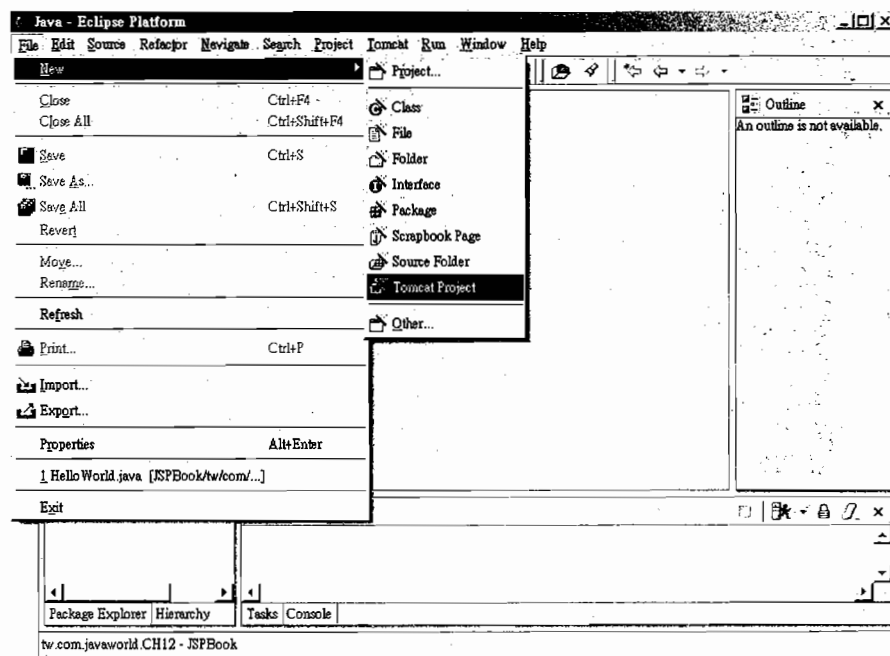


图 12-14 从菜单的【File | New】点击【Tomcat Project】

■ 使用 Eclipse 来启动和停止 Tomcat 的设定

打开菜单上的【window | preferences】，可以发现设定画面中出现 Tomcat 的选项。点击它并在这里设定 Tomcat Version, Tomcat home, Configuration file。另外底下的 Launch Tomcat using Security Manager 这个 check box 无须打勾，如图 12-15。因为这次使用的是 Tomcat 5.0.16，所以在这里笔者点击 version 5.x；另外将 Tomcat home 指到我们安装 Tomcat 的路径，笔者将 Tomcat 安装于 *C:\Program Files\Apache Software Foundation\Tomcat 5.0*，因此指到上述路径。设定完成后，按下【OK】就可以了。

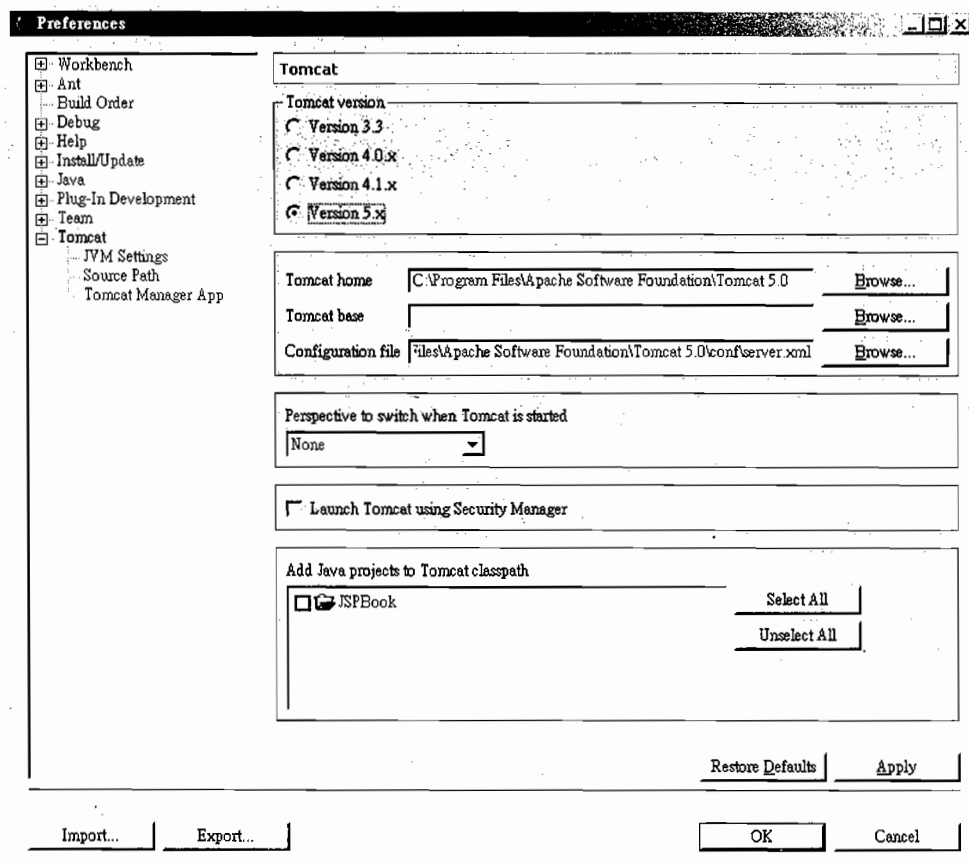




图 12-15 设定 Eclipse Preference -> Tomcat

设定完后可以发现工具栏上会有 Start Tomcat 的选项 ，点击后将会启动 Tomcat，并且会在 console 窗口出现信息。同样地，要停止 Tomcat 时，点击【Stop Tomcat 】将会停止 Tomcat，并将信息显示在 console 窗口中。

JSP2.0 技术手册

■ SolarEclipse Suite of Development Tools 的安装

SolarEclipse Suite of Development Tools 是用来当编辑 XML, JSP, CSS 时, 帮我们把 tag 和批注等加上颜色用的 plug-in。此外 plug-in 还有 Properties Editor 的 Unicode 编辑功能, 在建立内容为中文 *.property* 文件时, SolarEclipse 将会自动把中文转换成 Unicode 编码。

和上一个 plug-in 安装方式不同, SolarEclipse Suite of Development Tools 支持 Eclipse 的 update 功能, 因此可以利用 Eclipse 的 update manager 将 plug-in 所在的网址设定好后, 便会自动将 plug-in 安装好。当然也可以从网站上或光盘中获得压缩文件, 如果使用压缩文件, 则如同上一个 plug-in 的安装方式, 将其解压缩后放到 Eclipse 安装文件夹下的 *plugins* 中即可。

首先点击菜单的【Help | Software Updates | Update Manager】, 启动 Update Manager。启动后的画面如图 12-16:

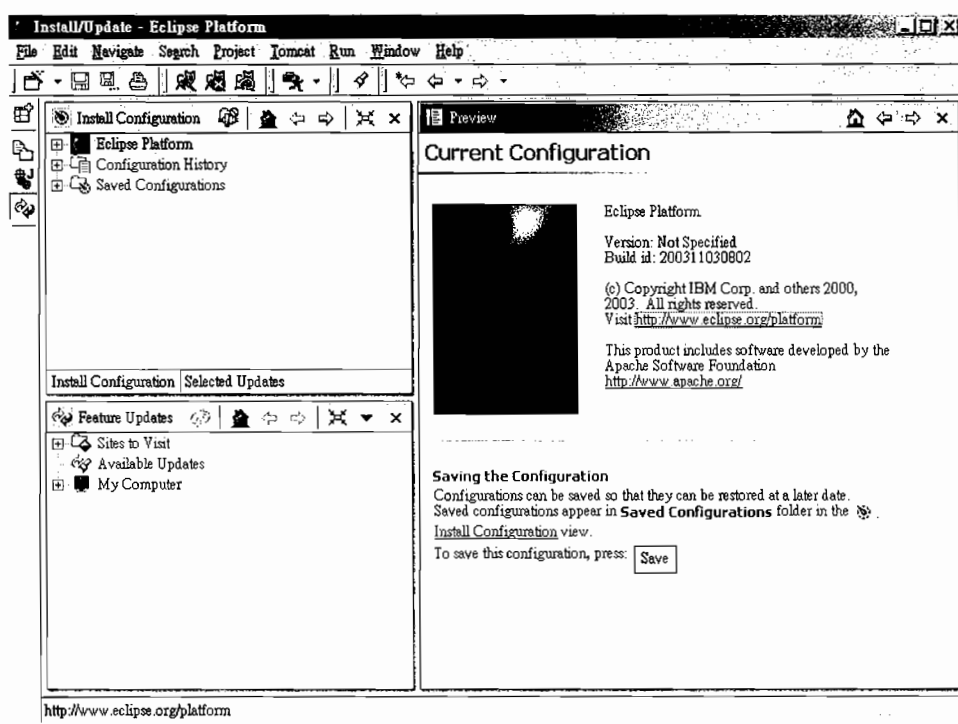


图 12-16 启动 Update Manager

在画面左下角的 Feature Updates 窗口中的 My Computer 按下右键后, 选择【New | Site Bookmark】。按下后将会出现如图 12-17 的画面, 请输入名称: SolarEclipse 和网址: <http://solareclipse.sourceforge.net/updates/>, 然后按下【Finish】。

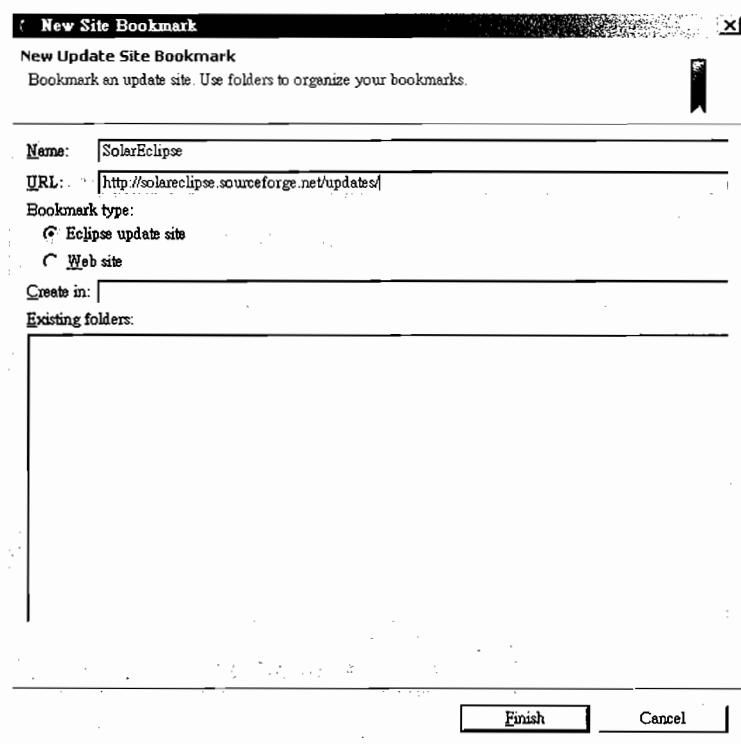


图 12-17 New Site Bookmark

设定完后，在左下角的 Feature Updates 窗口中会出现 SolarEclipse 的树状节点。可以从此树状节点中选择 SolarEclipse 0.4.0 并点击右边窗口中的【Install Now】安装(如图 12-18)。

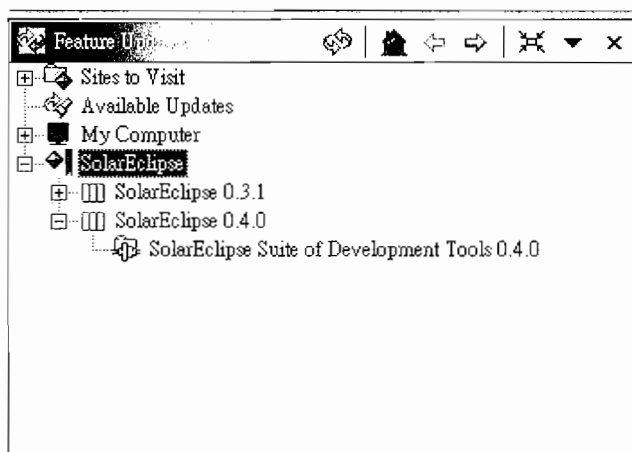


图 12-18 安装 SolarEclipse 0.4.0

注意

要设定 proxy, 选择菜单上的 Window | Preferences. 出现 Preferences 画面后, 选择【Install/Update】选项, 里面会有 Proxy Setting 可以设定。

安装完后, 我们要重新启动 Eclipse, 请选择【yes】并重新启动 Eclipse。重新启动以后就可以使用 SolarEclipse 的功能了。

■ SolarEclipse 的设定和使用

点击菜单中的【Window | Preferences】, 可以发现 Preferences 窗口中多了一个【SolarEclipse】的选项, 我们可以在此设定 JSP, CSS 和 XML 的颜色, 如图 12-19:

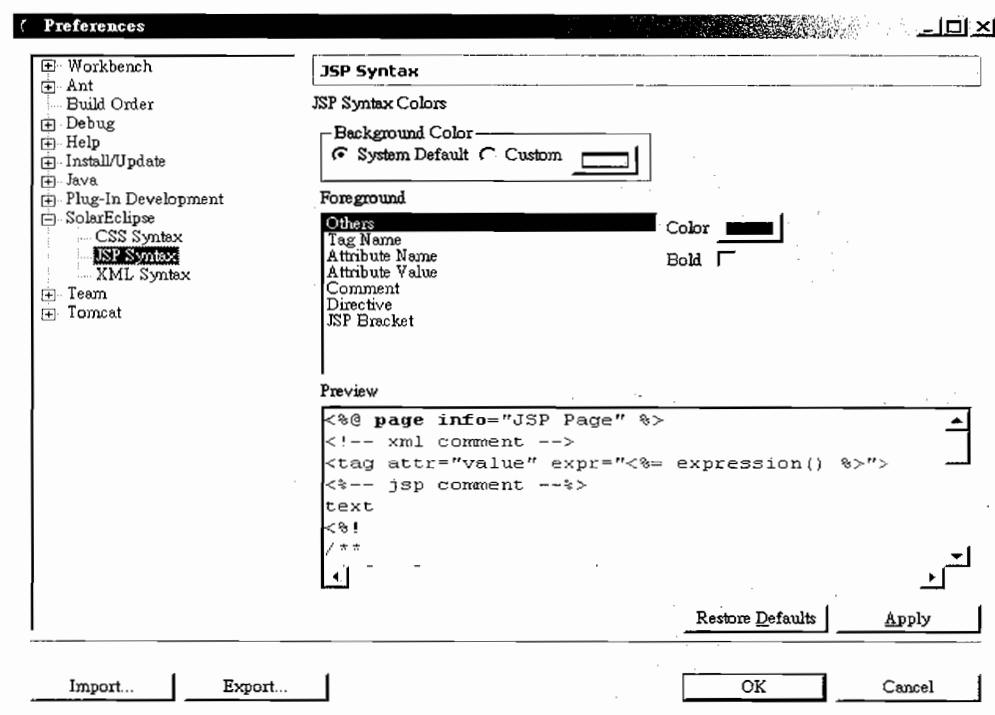


图 12-19 设定【SolarEclipse】

12-6 使用 Eclipse 来开发 Web Application(2)

在本节中笔者将介绍如何在 Eclipse 上开发 Servlet 与 JSP。

JSP2.0 技术手册

■ 新增 Tomcat project

首先选择菜单的【File | New | Tomcat Project】来新增一个 Tomcat Project。在设定画面中输入 Project 的名称，亦是 Web application 的名称。在这里笔者输入 JSPBookTomcat 作为我们 Web application 的名称，如图 12-20。

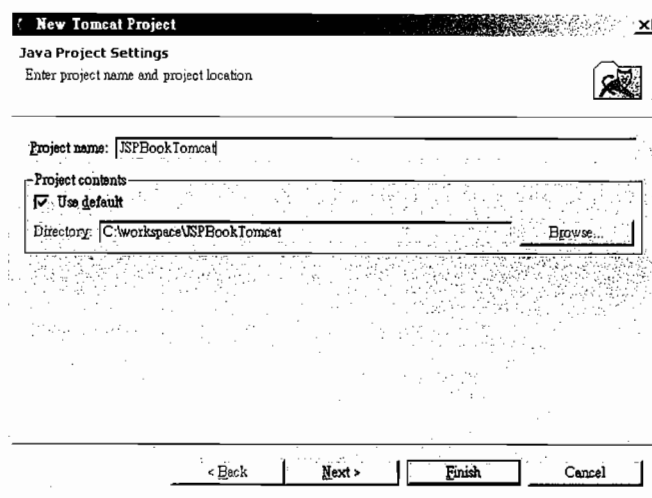


图 12-20 New Tomcat Project

点击【Next】到 Tomcat Project Settings 画面，在 Subdirectory to set as web application root(optional)的字段同样填上 JSPBookTomcat，并且将 *Can update server.xml* file 这个 check box 打勾，如图 12-21：

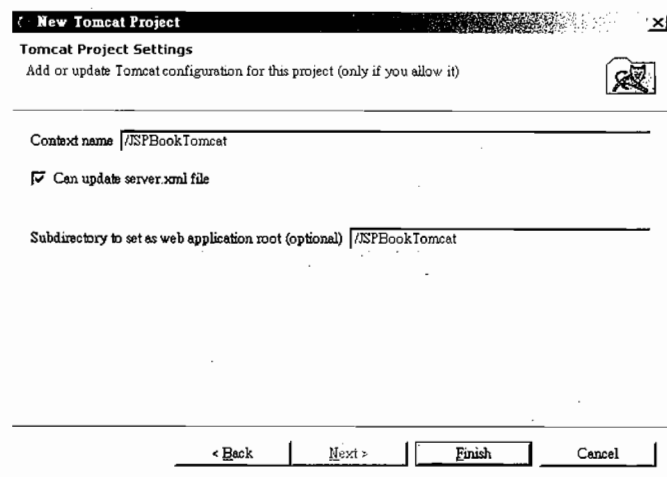


图 12-21 Tomcat Project Settings

设定完后，就可以发现主画面中出现 JSPBookTomcat 的 project。

■ 撰写 HelloWorldServlet.java 和 HelloWorld.jsp

本节中笔者将介绍如何撰写 JSP 和 Servlet。首先我们必须先新增一个 *web.xml*。点击 JSPBookTomcat project 中 *JSPBookTomcat* 下的 *WEB-INF*，按下右键选取【New File】，会产生图 12-22，文件名请填写为 *web.xml*。

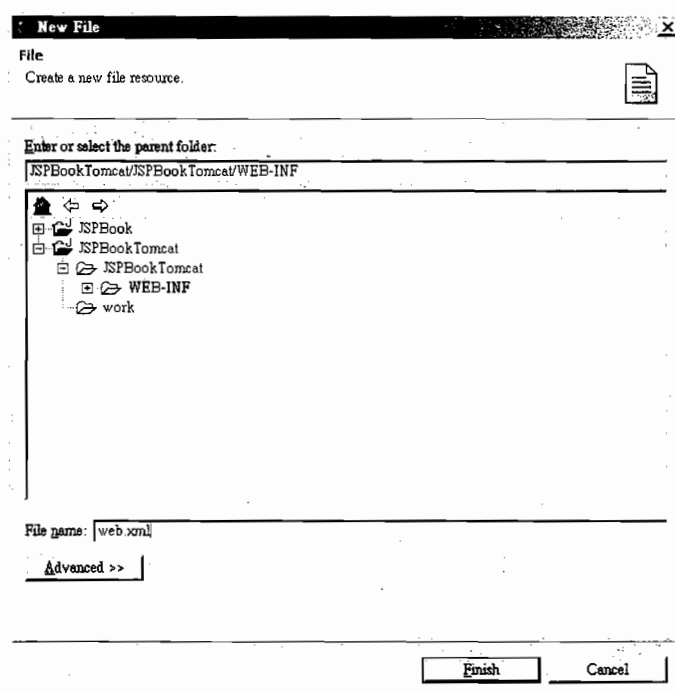


图 12-22 新增 web.xml

按下【确定】后会产生 *web.xml*，我们将 *web.xml* 设计如下：

■ web.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
  version="2.4">
  <display-name>JSPBook CH12</display-name>
  <description>
    JSPBook CH12 Example servlets and JSP pages.
  </description>
</web-app>
```

JSP2.0 技术手册

接下来就是撰写 *HelloWorldServlet.java*，在 JSPBookTomcat 按右键，选取【New | Class】，在 Package 栏中填入 *tw.com.javaworld.CH12*，Name 栏则填入 *HelloWorldServlet*，而在 Superclass 栏填入 *javax.servlet.http.HttpServlet*(如图 12-23)。设定完后按下【Finish】。

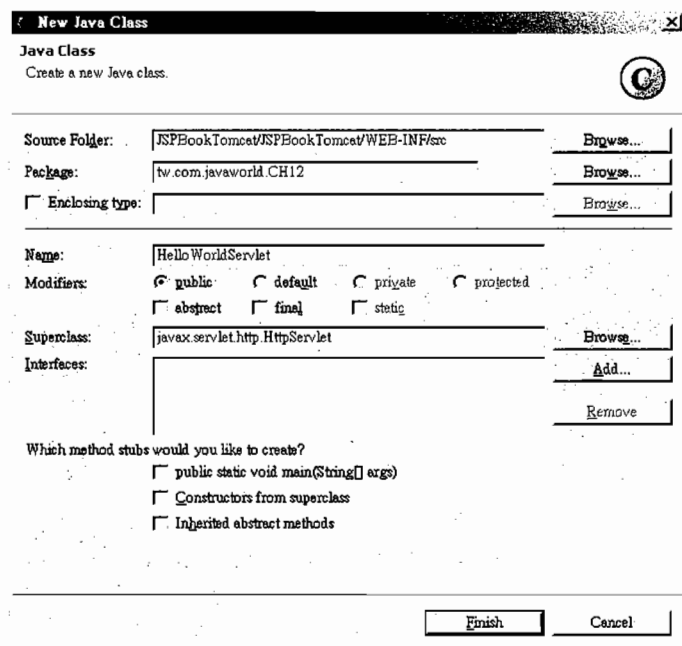


图 12-23 建立 HelloWorldServlet

HelloWorldServlet.java 程序代码如下：

```
■ HelloWorldServlet.java
/*
 * Created on 2003/11/3
 *
 * To change the template for this generated file go to
 * Window>Preferences>Java>Code Generation>Code and Comments
 */
package tw.com.javaworld.CH12;

import java.io.*;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
/**
 * @author morchory
 *
 * To change the template for this generated type comment go to
```

JSP2.0 技术手册


```

* Window>Preferences>Java>Code Generation>Code and Comments
*/

public class HelloWorldServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse
        response)
        throws ServletException, IOException {
        PrintWriter out = response.getWriter();
        out.println("Hello World from Eclipse!!");
    }
}

```

将上述 servlet 写完后, 接下来就是部署我们写的 servlet, 将 *web.xml* 改写如下:

■ web.xml

```

<?xml version="1.0" encoding="ISO-8859-1"?>

<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
    version="2.4">
    <display-name>JSPBook CH12</display-name>
    <description>
        JSPBook CH12 Example servlets and JSP pages.
    </description>
    <servlet>
        <servlet-name>HelloWorldServlet</servlet-name>

        <servlet-class>tw.com.javaworld.CH12.HelloWorldServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>HelloWorldServlet</servlet-name>
        <url-pattern>/HelloWorldServlet</url-pattern>
    </servlet-mapping>
</web-app>

```

完成后启动 Tomcat, 打开浏览器, 在地址栏打上 <http://localhost:8080/JSPBookTomcat/HelloWorldServlet> 产生图 12-24, 顺利完成在 Eclipse 上开发 Servlet。

接下来介绍在 Eclipse 上开发 JSP 程序 *HelloWorld.jsp*。首先在 JSPBookTomcat 下的 JSPBookTomcat 按右键, 选择【New | File】并将文件名设为 *HelloWorldJSP.jsp*, 如图 12-25。



图 12-24 执行 HelloWorldServlet

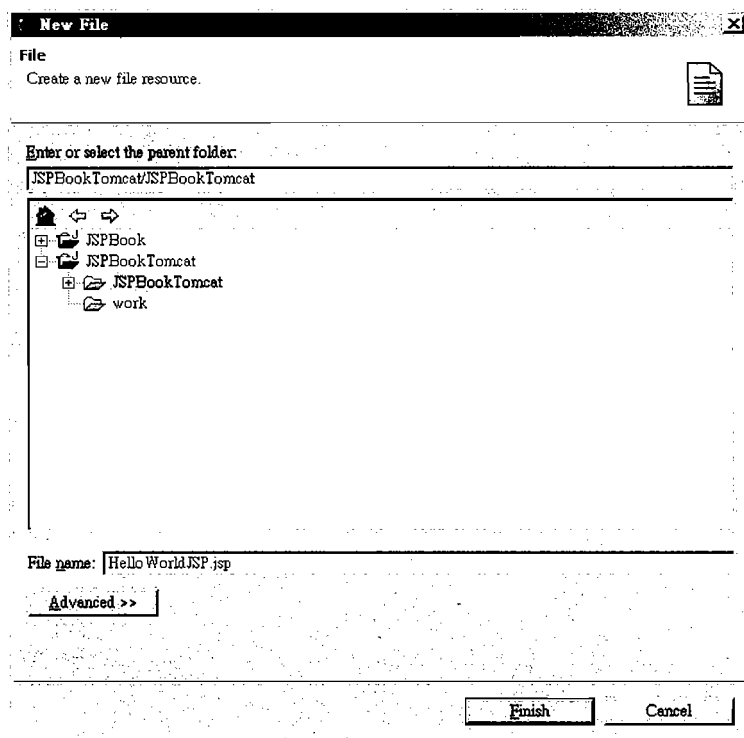


图 12-25 新增 HelloWorldJSP.jsp

HelloWorldJSP.jsp 程序内容如下：

■ HelloWorldJSP.jsp

```
<%@page language="java" contentType="text/html"%>
<%= "Hello World from Eclipse!!!" %>
```

启动 Tomcat 后，执行 <http://localhost:8080/JSPBookTomcat/HelloWorldJSP.jsp>。结果如图 12-26，顺利完成在 Eclipse 上开发 JSP。

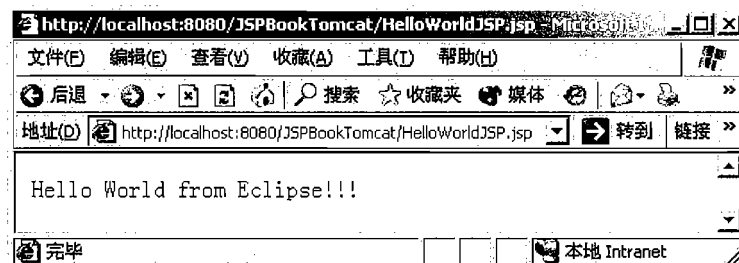


图 12-26 执行 HelloWorldJSP.jsp

JSP2.0 技术手册

13

第十三章

SQL 介绍

将数据存入数据库最大的好处是能对数据做分析，如：JSP、ASP、PHP 和 Perl 等等，依照不同的需求而呈现出不同的内容，大为减低开发的人力、财力和时间。例如：网站所提供的个性化服务，搜索引擎，在线订票等应用，皆以数据库作为后端架构所衍生出来的应用。

本章分 7 节来详细介绍 SQL 的功能和应用，内容如下：

- 13-1 数据库基本概念
- 13-2 SQL 简介
- 13-3 DDL 语句
- 13-4 数据的查询 —— SELECT
- 13-5 新增数据 —— INSERT
- 13-6 修改数据 —— UPDATE
- 13-7 删除数据 —— DELETE

JSP2.0 技术手册

13-1 数据库基本概念

假设现在我们有的一些顾客的相关数据，然后将它们储存在文件中，当数据量少时，可以直接解析里面的内容以取得所需的顾客数据，但是随着顾客人数的增加，数据量达到成千上万笔，若要找出任意条件的数据时，你可能会遇到性能低落以及一些操作的瓶颈。

有鉴于此，我们选择较好的解决方案，那便是使用数据库。数据库是将数据储存在一个特定的地方(如：表格)，以方便对数据做增加、删除、修改与查询的处理。数据库的最大特色在于：可以对特定的数据类型有着较好的搜索以及排序的算法，来管理不同类型的数据。因此通过数据库，数据可以得到有效的管理以及空间的分配。

数据库基本是以表格的形式将数据储存在一格一格的字段内，表格中某些行和列可以和另一个表格的行和列产生有意义的关联，而这就是关系型数据库命名的由来。将数据储存到数据库之后，就可以使用 SQL 语句对数据加以操纵，下面的章节将为读者介绍 SQL 的基本语法。

13-2 SQL 简介

SQL 全名为 Structured Query Language。ANSI（美国国家标准学会）声称，SQL 是关系数据库管理系统的标准语言。SQL 语句通常用于完成一些数据库的操作任务，比如：数据库中更新数据，或者从数据库中检索数据。常见关系数据库管理系统有：Oracle、Microsoft SQL Server、IBM DB2、MySQL 等等。绝大多数数据库系统除了能使用 SQL 之外，它们各自也有专属的扩展功能。然而一般常用的标准 SQL 命令，比如 Select、Insert、Update、Delete、Create 和 Drop 等等，在上述的数据库系统中皆可使用。

补充

SQL 不像其他的语言如：C、Pascal 等，它没有循环结构（比如 if-then-else、do-while）与函数定义等的功能。

依 SQL 的功能，可以概括为以下四组：

- 数据定义语言（Data Definition Language, DDL）
- 数据处理语言（Data Manipulation Language, DML）
- 数据控制语言（Data Control Language, DCL）
- 其他组成元素

上述的分组方式，只不过是依其所扮演的角色、定位与功能来加以分类的，并非绝对的分类型方式。笔者只是希望借此分类来让读者能够快速地了解 SQL。因此，接下来的章节将会依照上述的分组来一一讨论，但是顺序并非一致。

数据定义语言简称 DDL，它是用来定义与管理数据库及其所内含之各类对象的命令语句，

例如：建立、修改和删除数据库、数据表等对象的命令皆属于数据定义语言(DDL)。数据定义语言的格式如下列各项：

```
CREATE TABLE
ALTER TABLE
DROP TABLE
CREATE INDEX
```

在一般数据库默认状态下，只有数据库最高权限的成员才有权力去执行 DDL 命令，事实上，笔者也建议其他以外的账户都不应该有权去建立数据库对象，原因在于，如果不同的用户分别于数据库中建立各自拥有的数据库对象，则每一个对象的拥有者又必须将适当的权限赋给会存取对象的用户，如此恶性循环下来，将会使得管理作业变得非常的复杂，并且数据库的负担也相对加重，因此应尽量避免发生。此外，仅让这些成员拥有执行 DDL 命令的权限，也可以避免当对象的拥有者被从数据库中移除，造成对象拥有权的问题。

数据处理语言简称 DML，它是用来查询、新增、修改与删除数据库中数据的命令语句，下列命令皆属于数据处理语言：

```
SELECT: 用于查询数据
INSERT: 用于增加数据到数据库
UPDATE: 用于从数据库中修改现存的数据
DELETE: 用于从数据库中删除数据
```

数据控制语言简称 DCL，它是用来设定或变更数据库用户或角色之权限的命令语句。下面是几个数据控制语言的命令：

```
ALTER PASSWORD
GRANT
DENY
REVOKE
```

GRANT 命令能够赋予用户去存取某数据库对象或执行特定 SQL 指令的权限；DENY 命令能够否决用户的某项权限，并防止用户去继承其群组或成员的权限；REVOKE 命令能够将先前赋予或否决的权限移除。

除了上述三项外，还包括下列 SQL 相关的其他组成元素：

- 常数
- 运算符
- 函数

13-2-1 常数

常数(Constant)是一个固定的数据项或是一个代表特定数据值的符号，常数的格式依数据值的数据类型而有所不同。撰写程序的过程中，经常会使用到各种数据类型的常数，例如：假设我想要查询出生日期为 1978/12/27 的员工数据，此时您就必须学会如何于查询命令语句中表示

日期值为 1978/12/27。由此可知，了解各种数据类型之常数的表示格式是非常基本且重要的。一般常见的常数数据类型大约可以分为三类：数字类型、日期和时间类型、字符和字符串类型，下面用表 13-1、表 13-2、表 13-3 分别来介绍这几类类型：

注意

这里的类型是依 SQL-92、MySQL 为参考依据的。其中 SQL-92 为标准的类型，但是有些 MySQL 别名的类型是特属 MySQL 专用。因此，笔者建议实现时最好使用标准的 SQL-92 数据类型。

表 13-1 数字类型

SQL-92	MySQL 别名	容 量	说 明
	TINYINT	1 byte	-128 到 127
SMALLINT		2 byte	-32768 到 32767
	MEDIUMINT	3 byte	-8388608 到 8388607
INT、INTEGER		4 byte	-2147483648 到 2147483647
	BIGINT	8 byte	-9223372036854775808 到 9223372036854775807
	UNSIGNED TINYINT	1 byte	0 到 255
	UNSIGNED SMALLINT	2 byte	0 到 65535
	UNSIGNED MEDIUMINT	3 byte	0 到 16777215
	UNSIGNED INT	4 byte	0 到 4294967295
	UNSIGNED BIGINT	8 byte	0 到 18446744073709551615
NUMERIC NUMBER (PRECISION, SCALE)		8 byte	储存指定位数(PRECISION)之数值及固定位数之小数点位数(SCALE)。其中 PRECISION 默认值为 32；SCALE 默认值为 0。例如：NUMERIC(7,2)表示总长度为 7 位数，小数点位数为 2 位。
DECIMAL(PRECISION, SCALE)		8 byte	同 NUMERIC
FLOAT(PRECISION)		4 byte	-3.402823466E+38 到-1.175494351E-38，0 和 1.175494351E-38 到 3.402823466E+38
DOUBLE PRECISION		8 byte	1.7976931348623157E+308 到 -2.2250738585072014E-308、0 和 2.2250738585072014E-308 到 1.7976931348623157E+308
REAL		8 byte	同 DOUBLE PRECISION

表 13-2 日期和时间类型

SQL-92	MySQL 别名	容量	说 明
DATE		3 byte	'1000-01-01'到'9999-12-31'。'YYYY-MM-DD'格式来显示 DATE 值
TIME		3 byte	'-838:59:59'到'838:59:59'。'HH:MM:SS'格式来显示 TIME 值
TIMESTAMP		4 byte	'1970-01-01 00:00:00'到 2037-12-31 23:59:59'
	DATETIME	8 byte	'1000-01-01 00:00:00'到'9999-12-31 23:59:59'。'YYYY-MM-DD HH:MM:SS' 格式来显示 DATETIME 值
	YEAR	2 byte	1901 到 2155 (4 位年格式); 1970-2069 (70-69) 2 位年格式。默认为 4 位

表 13-3 字符和字符串类型

SQL-92	MySQL 别名	数据储存范围	说 明
CHARACTER、CHAR		1~255 个字符	固定长度
CHARACTER VARYING、CHAR VARYING、VARCHAR		1~255 个字符	可变长度
	TINYTEXT	1~255 个字符	
	TINYBLOB	1~255 个字符	储存二进制数据
	TEXT	1~65535 个字符	
	BLOB	1~65535 个字符	储存二进制数据
	MEDIUMTEXT	1~16777215 个字符	
	MEDIUMBLOB	1~16777215 个字符	储存二进制数据
	LONGTEXT	1~4294967295 个字符	
	LOBLOB	1~4294967295 个字符	储存二进制数据

所有常数中，**数字类型**的格式最多，限制也最多。当不同的数据库在做交换数据时，数值的精确度也最容易降低，例如：Oracle 和 MS SQL 之间会发生同样的数字类型，但是它能储存的长度却不一样，导致它们之间的数据在传递过程中，会截短数字，改变它们的数值。因此在移植前，程序员必须要了解两者平台间的数据差异，以及数据精确度的风险。

所有的数值都有精确度，精确度指的是：有效数字位数和小数点。例如，数字 1234.56 的精确度为 6，小数点位数为 2，可以定义为 NUMERIC(6,2)。

除了获取数值长度和其他数值处理所需的属性外，SQL-92 还提供了内建函数，如：加、减、乘、除等等，而且所有的数值也都可以互相比较。

以上就是三种常数数据类型：数字类型、日期和时间类型、字符和字符串类型，接下来为读者介绍运算符。

13-2-2 运算符

运算符(Operators)是一些特殊的符号，它们能够用来执行算术运算、字符串连接、赋值数据值以及在字段、常数与变量之间进行比对。本小节我们将讨论各种类型的运算符以及运算时的优先级。

一般而言，运算符通常分为下列六类：

- 1. 算术运算符 (Arithmetic Operators)
- 2. 赋值运算符 (Assignment Operators)
- 3. 位运算符 (Bitwise Operators)
- 4. 比较运算符 (Comparison Operators)
- 5. 逻辑运算符 (Logical Operators)
- 6. 字符串连接运算符 (String Concatenation Operators)

笔者现在就依序介绍这六大类运算符。

算术运算符(Arithmetic Operators)通常用在普通的计算。算术运算符适用于各种数字类型的数据，它与我们使用的四则运算几乎相同，格式或括号的使用方法与普通的算法没什么不同。算术运算符的种类如表 13-4 所示：

表 13-4

运算符	符号	例子	说明
加法	+	X + Y	X 加上 Y
减法	-	X - Y	X 减去 Y
乘法	*	X * Y	X 乘上 Y
除法	/	X / Y	X 除以 Y
余数	%	X % Y	X 除以 Y 的余数

算术运算符绝对适用于各种数字类型的数据，但是请读者注意，我们一样也可以使用算术运算符“+”与“-”替 datetime 或 smalldatetime 数据类型的日期时间值加上或减去一个数值，在这种状况下，此数值的单位将是天数(亦即 24 小时，或是 1440 分钟，亦或是 86400 秒)。假若替日期时间值加上或减去的数值包括小数，就必须把小数部分换算成小时、分钟或秒。例如：我们替日期时间 "01/01/1999 13:00:00" 加上 10.55，因为 10.55 等于 10 天 + 0.55 天，而 0.55 天又等于 0.55 * 24 小时 = 13.2 小时，而 0.2 小时又等于 0.2 * 60 分钟 = 12 分钟，因此最后得到的结果将是 "01/12/1999 02:12:00"。

赋值运算符(Assignment Operators)只有一个“等号(=)”。我们可以用赋值运算符将数据值赋值给特定的对象，比方说，在 SELECT 命令语句中可以使用赋值运算符将数据赋值给局部变量。此外，我们亦可在 SELECT 命令语句中使用赋值运算符来自定义查询结果的字段标题。以下列

的程序代码而言，我们将查询结果的第一个与第二个字段标题分别设定为 "员工姓名" 与 "年龄"：

```
SELECT 员工姓名 = FirstName + ' ' + LastName, 年龄 = 24 FROM Employees
```

位运算符(Bitwise Operators)有四个运算符，分别为 &、|、~、^，如表 13-5 所示：

表 13-5

运算符	说 明
&	位转换 AND
	位转换 OR
~	位转换 NOT
^	位转换互斥性 OR

位运算符能够在整数数据或二进制数据(IMAGE 数据类型除外)之间执行位处理，但是大家必须注意，在位运算符左右两侧的操作数不能同时为二进制数据，从下列表中可以清楚地发现，当左侧的操作数为特定数据类型的数据时，右侧的操作数所允许的数据类型（见表 13-6）。

表 13-6

左侧操作数	右侧操作数
binary	int、smallint 或 tinyint
bit	int、smallint、tinyint 或 bit
int	int、smallint、tinyint、binary 或 varbinary
smallint	int、smallint、tinyint、binary 或 varbinary
tinyint	int、smallint、tinyint、binary 或 varbinary
varbinary	int、smallint 或 tinyint

比较运算符(Comparison Operators)有八个运算符，如表 13-7 所示：

表 13-7

运算符	符 号	例 子	说 明
大于	>	X > 10	X 是否大于 10
小于	<	X < 10	X 是否小于 10
等于	=	X = 10	X 是否等于 10
不等于	<>	X <> 10	X 是否不等于 10
大于等于	>=	X >= 10	X 是否大于等于 10
小于等于	<=	X <= 10	X 是否小于等于 10
不大于	!>	X !> 10	X 是否不大于 10
不小于	!<	X !< 10	X 是否不小于 10

比较运算符可以用来检测两个操作数是否相同，比较的结果为布尔值：TRUE(表示两个操作数相同)、FALSE(表示两个操作数不相同)或 UNKNOWN。一般来说，会传回布尔值的表达式即是布尔表达式，亦称为条件式。我们最常在 WHERE 或 HAVING 自变量中使用布尔表达式来限定仅有符合特定条件的数据才会加以处理。

逻辑运算符(Logical Operators)包括 AND、OR 与 NOT 等三个逻辑运算符，它的主要功能是让用户能在布尔表达式中进行 AND(且)、OR(或)与 NOT(否定)的运算。有时候也会把 ALL、ANY、BETWEEN、EXISTS、IN、LIKE 与 SOME 等运算符归为逻辑运算符之中，但是由于这几个运算符主要使用于 SELECT 命令的 WHERE 自变量中，因此笔者将留至说明查询命令 SELECT 时再加以说明。

字符串连接运算符(String Concatenation Operators)就是“加号(+)”，它可以将多个字符串常数合并成一个字符串常数。以下面的命令语句而言，所得的结果将是 'JSP 2.0 技术手册'：

```
SELECT 'JSP 2.0' + '技术手册'
```

当 SELECT 查询数据表时，你可以使用“加号(+)”将 char、varchar 等数据类型的字段合并在一起，它们甚至可以与字符串常数合并在一起，如下范例所示：

```
SELECT '姓名：' + Name FROM Employees WHERE Birthday = 27
```

以上六大类运算符简单介绍到此，接下来介绍的内容也是和运算符有关的，那就是运算符的优先级。

当一个复杂的表达式拥有多个运算符时，运算符的优先级将决定哪一个运算会先被执行。因此，运算符的优先级对运算结果有绝对的影响力。一般而言，运算符的优先级分成好几个等级，等级较高者将比等级较低者先被执行。运算符的等级由高至低如下所示：

+ (正号)、- (负号) 与 ~(位转换 NOT)
* (乘)、/ (除) 与 % (余数)
+ (加)、+ (字符串连接运算符) 与 - (减)
=、>、<、>=、<=、<>、!>、!<
^ (位转换互斥性 OR)、& (位转换 AND) 与 ! (位转换 OR)
NOT
AND
ALL、ANY、BETWEEN、IN、LIKE、OR 与 SOME
赋值运算符(=)

这里有个地方需要留意的，如果表达式中有两个同等级的运算符，将会依照它们在表达式中的位置由左至右执行。运算符的部分，笔者就介绍到此，最后要介绍的是函数(function)。

13-2-3 函数

函数(function)主要用来执行一些特殊的运算,方便用户处理数据。每一个函数语句皆包含一个名称,名称之后紧接的是一对小括号,例如:AVG()。大部分的函数在此对小括号中需要一个或一个以上的参数。

若依照使用类型来区分,SQL-92 标准的函数可分为下列两类:

■ 汇总函数(Aggregate Function)

汇总函数主要使用于 SELECT 命令的选取行中,它们能够传回数据记录的一些统计信息。以下的命令语句而言,我们使用汇总函数 **AVG()** 来计算员工的平均薪资:

```
SELECT AVG(Salary) FROM Employees
```

SQL-92 标准的汇总函数共有五个,它们分别为:AVG()、COUNT()、MAX()、MIN() 和 SUM(),读者直接看表 13-8,就能清楚它们分别代表的意思为何:

表 13-8

汇总函数	范 例	说 明
AVG()	AVG(Salary)	计算出取得薪水数据的薪资平均数
COUNT()	COUNT(Salary)	计算出取得薪水数据的数据笔数
MAX()	MAX(Salary)	计算出取得薪水数据中的最大值
MIN()	MIN(Salary)	计算出取得薪水数据中的最小值
SUM()	SUM(Salary)	计算出取得薪水数据的薪资总合

这部分留在说明查询(SELECT)命令时再做更详细的说明。

■ 纯量函数(Scalar Function)

纯量函数通常是针对传递给它的一个或多个参数值来运算处理的,并传回一个单一值。纯量函数可以使用在任一表达式中,对用户来说是十分方便的。若依功能特性来区分,纯量函数可区分为四类(见表 13-9):

表 13-9

函数类	说 明
字符串函数	字符串函数能够用来处理 char、varchar 等等数据类型
日期时间函数	日期时间函数用来处理日期与时间值
算术函数	算术函数能帮助我们完成几何运算、三角函数运算与各种一般的运算
中继数据函数	中继数据函数能够传回数据库与数据库对象之特定属性的信息。比方说,如果你想要知道目前哪一个数据库是作用数据库,只须借助中继数据函数 DB_NAME()即可,例如:SELECT DB_NAME()

SQL-92 标准主要就是上述四大类,但是有些数据库厂商,如: Microsoft 的 SQL Server, 它的函数功能除了包含 SQL-92 标准四大类之外,还拥有其他功能的函数,例如: 安全性函数、系统函数、系统统计函数、文本与影像函数等等。因此,读者别以为只有上述四类函数,还必须参看读者使用的数据库,至于此部分只好由读者自行参考数据库的文件说明。

■ 字符串函数(String Function)

下列为 SQL-92 标准所提供的字符串函数,我们可以使用这些函数对 char、varchar 等数据类型的数据进行各种不同的处理,并回传字符串数据处理后的数据值(见表 13-10)。

表 13-10

ASCII()	LENGTH()	REPLACE()	SUFFIX()
CHAR()	LOWER()	RIGHT()	SUBSTRING()
LEFT()	LTRIM()	RTRIM()	UPPER()

以下我们将介绍一些基本且使用频率极高的字符串函数。

● LENGTH()

假若我们想得知某一字符串的长度,可以使用 LENGTH()函数:

LENGTH(char_expression)

LENGTH()函数能传回字符串表达式 char_expression 的长度。这里必须注意一点: LENGTH()函数所传回的数值是字符数而不是字节数,而且它会先去除字符串表达式的尾部空白,再传回其值。如果字符串表达式是一个空字符串,LENGTH()函数将传回数值为零。

范例:

```
SELECT LENGTH('JSP 技术手册')           → 传回 7
SELECT LENGTH('JSP 技术手册  ')         → 传回 7
SELECT LENGTH('')                        → 传回 0
```

● SUBSTR()

SUBSTR()函数能够自 char、varchar、binary、varbinary 等数据类型的数据中抽取某一部分的字符,其语法如下所示:

SUBSTR(char_expression, start_pos, length)

SUBSTR()函数能够从 char_expression 参数所指定的表达式中,自 start_pos 参数所指定的位置开始,抽取 length 参数所指定个数的字符。以下面的命令语句而言,表示将抽取字符串 'JavaServer Page' 第十二个字符开始算起的四个字符:

```
SELECT SUBSTR('JavaServer Page', 12, 4) → 传回字符串 Page
```

JSP2.0 技术手册

● LEFT()

LEFT()函数可以抽取字符串左侧算起特定个数的字符,语法如下:

LEFT(char_expression, count)

LEFT()函数会从字符串表达式 char_expression 最左边的字符开始算起,抽取 count 参数所指定个数的字符。

```
SELECT LEFT(name,4) FROM user WHERE name = 'Goldman';    ➔ 传回字符串 Gold
```

● RIGHT()

RIGHT()函数可以抽取字符串右侧算起特定个数的字符,语法如下:

RIGHT(char_expression, count)

RIGHT()函数会从字符串表达式 char_expression 最右边的字符开始算起,抽取 count 参数所指定个数的字符。

```
SELECT RIGHT(name,4) FROM user WHERE name = 'Goldman';    ➔ 传回字符串 dman
```

注意

LEFT()和 RIGHT()的 count 代表字符数而不是字节数,而且即使 char_expression 内含中文字,中文字亦被视为占用 1 个字符位置,而不是 2 个字符位置。

● LTRIM()

LTRIM()函数可以移除字符串的前置空白,语法如下:

LTRIM(char_expression)

```
LTRIM(' JSP 技术手册')    ➔ 传回字符串 JSP 技术手册
```

● RTRIM()

RTRIM()函数可以移除字符串的尾部空白,语法如下:

RTRIM(char_expression)

RTRIM()函数可以移除字符串表达式的尾部空白,如下范例:

```
RTRIM('JSP 技术手册')    ➔ 传回字符串 JSP 技术手册
```

● LOWER()

LOWER()函数可以将字符串中所有大写字母转换成小写字母,语法如下:

LOWER(char_expression)

你可以使用 LOWER()函数,将字符串表达式中所有大写字母转换成小写字母,但是

JSP2.0 技术手册

LOWER()函数并不会影响非英文字母的字符, 如下:

```
SELECT LOWER('JSP 技术手册')    ➔ 传回字符串 jsp 技术手册
```

● UPPER()

UPPER()函数可以将字符串中所有小写字母转换成大写字母, 语法如下:

UPPER(char_expression)

你可以使用 UPPER()函数将字符串表达式中的所有小写字母转换成大写字母, 但是 UPPER()函数并不会影响非英文字母的字符, 如下:

```
SELECT UPPER('jsp 技术手册')    ➔ 传回字符串 JSP 技术手册
```

■ 日期时间函数

日期时间函数可以让 datetime 与 smalldatetime 数据类型的数据进行各种不同的运算处理, 并回传一个字符串、数值或日期时间值, 表 3-11 为 SQL-92 标准所提供的日期时间函数:

表 3-11

ADD_MONTHS()	DAYOFMONTH()	MINUTE()	NEXT_DAY()
CURDATE()	DAYOFWEEK()	MONTH()	SECOND()
CURTIME()	DAYOFYEAR()	MONTHNAME()	WEEK()
DAYNAME()	HOUR()	MONTHS_BETWEEN()	YEAR()

● CURDATE()、CURTIME()

CURDATE()和 CURTIME()函数都不具有任何参数, 它们都是回传系统目前的日期和时间。

```
SELECT CURDATE()
```

● YEAR()、DAYOFYEAR()

YEAR()和 DAYOFYEAR()函数的语法如下所示:

YEAR(date_expression)

DAYOFYEAR(date_expression)

YEAR()和 DAYOFYEAR()函数都能够取得日期时间表达式 date_expression 中的公元年份数值, 它们都固定传回四位数的公元年份值。

```
SELECT YEAR('01/28/2003')    ➔ 传回 2003
```

```
SELECT DAYOFYEAR('01/28/2003 3:20:50 PM')    ➔ 传回 2003
```

JSP2.0 技术手册

● MONTH()、DAYOFMONTH()

MONTH()和DAYOFMONTH()函数的语法如下所示:

```
MONTH(date_expression)
DAYOFMONTH(date_expression)
```

MONTH()和DAYOFMONTH()函数都能够取得日期时间表达式 date_expression 中的月份数值, 传回的月份是以数值 1~12 来表示的, 1 代表一月、2 代表二月, 其余依此类推。

```
SELECT MONTH ('01/28/2003')          → 传回 1
SELECT DAYOFMONTH ('01/28/2003 3:20:50 PM') → 传回 1
```

■ 算术函数

算术函数可以对 decimal、float、real、int、smallint、tinyint 等数据类型的数值表达式进行各种不同的运算, 并传回计算结果。使用算术函数时, 必须注意到 ABS()、CEILING()、DEGREES()、FLOOR()、POWER()、RADIANS()与 SIGN()所传回的数值类型将与其参数的数据类型相同; 然而 ACOS()、COS()、ATAN()、ATAN2()、COT()、TAN()、SIN()、ASIN()、EXP()、LOG10()、SQUARE()与 SQRT()等算术函数则会自动将其参数之数值表达式转换成 float 数据类型, 且回传的数值亦为 float 数据类型。

由于算术函数非常容易了解与使用, 因此笔者仅对一些常用算术函数的功能简述于表 13-12 中:

表 13-12

算术函数	功能简述
ABS()	传回数值表达式的绝对值
ACOS()	传回数值表达式的反余弦弧度值
ASIN()	传回数值表达式的反正弦弧度值
ATAN()	传回数值表达式的反正切弧度值
ATAN2()	传回四个象限的反正切弧度值
CEILING()	传回大于或等于所指定之数值表达式的最小整数
COS()	传回数值表达式的余弦值
DEGREES()	将某一个数值表达式由单位为“弧度(radians)”转换成单位为“度(degree)”
EXP()	传回 e 的 N 次方之值
FLOOR()	传回小于或等于所指定之数值表达式的最大整数
LOG10()	以 10 为底数来计算所指定之数值表达式的对数值
PI()	传回圆周率的值
POWER()	传回所指定之数值表达式的特定次方之值
RADIANS()	将某一个数值表达式由单位“度(degree)”转换成单位为“弧度(radians)”
RAND()	传回介于 0~1 之间的一个随机数值

续表

算术函数	功能简述
ROUND()	依指定的小数字数来将一数值表达式四舍五入
SIGN()	根据指定数值表达式的“数学符号”，SIGN()函数将传回 1(表示正数)、-1(表示负数)或 0(表示为 0)
SIN()	传回数值表达式的正弦值
SQRT()	传回数值表达式的平方根
TAN()	传回数值表达式的正切值

以上是三个其他 SQL 相关的组成元素(常数、运算符、函数)介绍，在接下来一节中，笔者将为读者介绍 SQL 最重要的两个组成元素：DML 和 DDL。

13-3 DDL 语句

DDL 语句主要用来新增、删除数据库，和新增、修改、删除数据表格；而 DML 用于新增、搜索、修改、删除数据等等，接下来就先介绍 DDL 语句。

13-3-1 新增、删除数据库

当我们要使用数据库时，首先建立一个空的数据库，然后再依照需求新增数据表格。新增数据库的语句和删除的语句很相似，也很简单，不过在使用删除数据库的语句时，必须要小心谨慎，因为它可能会把数据库内所有的数据完全清除干净。新增、删除数据库的语句，如下所示：

■ 新增、删除数据库

```
CREATE DATABASE db_name
DROP DATABASE db_name
```

从字面上来看，可以很清楚地知道 CREATE DATABASE 是新增数据库的语句，而 db_name 则是你想命名的数据库名称；同样地，DROP DATABASE 是删除数据库的语句，db_name 则是数据库名称。这里有一个很重要的概念，SQL 语句中是不分大小写的，因此 CREATE DATABASE 若改为 create database 一样都是合法的 SQL 语句。

13-3-2 新增、修改、删除数据表格

当我们新增数据库后，接下来就是建立所需要的数据表格。一般基本的新增数据表格语句如下：

■ 新增数据表

```
CREATE TABLE tablename ( column1 data1 type,
                           column2 data2 type,
                           .....
                           .....
                           columnN dataN type,);
```

另外还有一种包含限制(constraint)的语句, 如下:

```
CREATE TABLE tablename ( column1 data1 type [constraint],
                           column2 data2 type [constraint],
                           .....
                           .....
                           columnN dataN type [constraint]);
```

为了让读者能够快速地了解, 下面笔者将举个简单的例子来说明:

```
CREATE TABLE employee ( first varchar(15),
                           last varchar(20),
                           age number(3),
                           address varchar(30),
                           city varchar(20),
                           state varchar(20));
```

建立一个新的数据表格时, 你可以在关键字 **CREATE TABLE** 之后命名数据表的名字, 然后是左括号 “(”, 接着第一个字段的名字, 再来是这一字段的数据类型, 再是任意的限制规则, 最后是右括号 “)”。你还要保证每一行定义之间有逗号分隔, 最后在 SQL 语句结束时, 加上分号 “;”。

表格和字段名称必须以字母开头, 第二个字符开始可以是字母、数字或者下画线, 但是字段名称的总长度不要超过 30 个字符。在定义表格和字段名称时, 千万不可使用 SQL 默认用于表格或者字段的关键词, 例如: **select**、**create**、**insert** 等等, 以避免错误的发生。

如果一个字段的名称为 “**Last_Name**”, 它是用来存放人名的, 所以字段类型就应该采用 “**varchar**” (variable-length character, 可变长度的字符)数据类型。

接下来, 我们来说明限制(constraint)。什么是限制呢? 所谓限制是规范字段的基本准则, 存入的数据都必须遵循这个准则, 否则将无法合法存入数据。下面举个例子:

```
CREATE TABLE employee ( id varchar(12) PRIMARY KEY,
                           first varchar(15) NOT NULL,
                           last varchar(20) NOT NULL,
                           age number(3),
                           address varchar(30),
                           city varchar(20),
                           country varchar(20) DEFAULT 'TW');
```

上述范例的意思是说: 我们建立一个数据表名为 **employee**, 它包含七个字段, 有 **id**、**first**、**last**、**age**、**address**、**city** 和 **country**。其中 **id** 设为 **PRIMARY KEY**(主键), 即设定 **id** 字段为

JSP2.0 技术手册

主索引字段，表示每一笔数据的惟一标识；再来设定 first 和 last 字段都为 NOT NULL，即设定此字段必须都有数据，不能为空白；最后又设定 country 的默认值为 tw，表示如果没有数据存入时，country 的值自动为 tw。

最后笔者再对一些常用的限制关键字用表 13-13 来说明：

表 13-13

关键字	说 明
PRIMARY KEY	设定该字段为主索引字段。此字段一定要有数据，且数据要惟一，不可重复
AUTO_INCREMENT	设定此字段为自动编号字段
DEFAULT value	默认 value 值给此字段
NULL	允许字段没有数据
NOT NULL	不允许字段没有数据

以上介绍 CREATE TABLE 建立数据表的语句告一段落，接下来笔者将介绍此节另外两个主题：修改数据表和删除数据表。

修改数据表的语句如下：

```
ALTER TABLE tablename ADD column data type [constraint] AFTER column1
ALTER TABLE tablename CHANGE old_column new_column new_data type
[constraint]
ALTER TABLE tablename DROP column
ALTER TABLE old_table RENAME new_table
```

所谓的修改数据表，除了包含修改字段名称、类型之外，还包括新增字段、删除字段、重新命名数据表名称等等。读者可以发现笔者在举例 ALTER TABLE 的语句时，共介绍正四种 ALTER TABLE 的用法。下面分别简单说明上述四个语句所代表的含意。

第一，在 tablename 的 column1 之后新增一个叫做 column 的字段，并且设定它的类型和限制规范。

第二，修改 tablename 中的 old_column 字段，并且重新命名为 new_column、类型为 new_data type 和其相关限制规范。其中 old_column 和 new_column 可以一致。

第三，删除 tablename 数据表格中的 column 字段。

第四，将原本名为 old_table 的数据表改名为 new_table。

最后介绍的是删除数据表，它的语句很简单，如下：

```
DROP TABLE tablename
```

删除名称为 tablename 的数据表。这里的 DROP TABLE 和上述 DROP column 不同，一个是删除数据表，连数据表内的所有字段都删除掉；另一个只是删除数据表中的某一字段。

到此为止，笔者已经介绍完常用到的 DDL 语言（Data Definition Language），其中包括

CREATE DATABASE、DROP DATABASE、CREATE TABLE、ALTER TABLE 和 DROP TABLE。

接下来为各位介绍 DML 语言 (Data Manipulation Language)，其中包括 SELECT、INSERT、DELETE、UPDATE 等等。其中 SELECT 是最常用，且功能最多、最复杂的，因此笔者在介绍 SELECT 时，会将它独立成一个章节来介绍说明；至于其他 INSERT、DELETE 和 UPDATE 则另为一个章节。

13-4 数据的查询——SELECT

数据库系统最基本且常见的作业，除了取得符合特定条件的数据记录外，还可以针对这些数据记录做更进一步的汇总、统计与分析，而这些作业可统称为【查询】(Query)。下列都是查询数据所常见的作业：

- 查询全体员工中哪一位员工的薪资最高
- 查询全体员工中哪一位员工的薪资最低
- 查询全体员工中薪资最高的前 20 位员工
- 查询全体员工中住台北市的有几人
- 查询哪一位员工的资历最长

数据查询可算是数据库最仰赖的功能之一，而且所有的数据查询作业完全由 SELECT 命令独立完成。

SELECT 命令语句可简可繁，命令行的长度依据查询的复杂度而定，它能够包含数据字段、常数值、变量、表达式或函数。

SELECT 命令的语句非常复杂，其主要自变量如下：

```
SELECT [DISTINCT | ALL] select_expression
FROM table_references
[INTO {OUTFILE | DUMPFILE} 'file_name' export_options]
[WHERE where_definition]
[GROUP BY col_name,...]
[HAVING where_definition]
[ORDER BY {unsigned_integer | col_name | formula} [ASC | DESC] ,...]
```

接下来笔者将说明 SELECT 命令的各个组成部分。

13-4-1 基本 SELECT 查询

首先我们来看一个最基本的 SELECT 查询：

```
SELECT name, phone_number, address FROM employee
```

上面的范例是最简单的 SELECT 查询，我们可以从这个范例得知 SELECT 命令至少必须

JSP2.0 技术手册

包含下列两个自变量：

- 将要查询的数据域位名称，即上述中的 name 或 phone 等；
- 查询的数据字段是来自哪个数据表，即语句中的 table_references。

SELECT 命令的数据区域中，各个字段的先后次序就是这些字段出现在查询结果中的先后次序。以上述范例而言，查询结果会依 name | phone_number | address 的顺序显示。

select_expression 除了可以是 FROM 自变量所指定数据表中的一个或多个字段之外，它还可以是下列两个项目：

- 由字段、常数与函数所组成的表达式，例如：

```
SELECT FirstName+LastName , YEAR(CURDATE()) FROM employee
```

- 万用字符 - 星号(*)。假若您要查询出所有字段的数据，只须使用一个星号(*)即可，因为在 SQL-92 标准中，星号是一个万用字符，表示所有的字段。下面范例是查询 employee 数据表中的所有字段数据：

```
SELECT * FROM employee
```

FROM 自变量表示在 SELECT 命令查询的过程中使用哪些数据表。

注意

如果读者在 FROM 自变量中所指定的数据表并非是作用数据库的数据表，请务必以 database_name.table_name 的格式来指定数据表。

13-4-2 关键字 ALL 与 DISTINCT 的使用

当我们使用 SELECT 命令来执行查询时，所有被查询的字段数据都会显示出来。但是，假若您不想显示重复的数据记录(即若有多笔相同的数据记录时，只会显示其中一笔，请加上关键字 DISTINCT。

注意

NULL 亦将被视为是相等的。

为了能够更清楚地了解关键字 ALL 与 DISTINCT，笔者利用范例来说明：

假若 employee 数据表中有六名员工，他们的性别(Sex)和年龄(Age)分别为：

性别(Sex)	M	F	M	M	M	F
年龄(Age)	25	24	25	31	33	31

当执行下列三道 SELECT 命令时，它们的结果分别为：

```
SELECT Age FROM employee      → 传回 25, 24, 25, 31, 33, 31
SELECT ALL Age FROM employee  → 传回 25, 24, 25, 31, 33, 31
```

```
SELECT DISTINCT Age FROM employee
```

 → 传回 25, 24, 31, 33

从上述范例的执行结果可以清楚地发现：当我们要显示所有数据时，可以加或不加关键字 ALL，因为 SELECT 命令的默认值为 ALL；假若我们不想显示重复的数据记录时，就加上关键字 DISTINCT 即可。

每一条 SELECT 命令仅能有一个 DISTINCT 关键字，这意味着关键字 DISTINCT 是限制所指定的字段数据都重复者，只显示其中一笔，而不是针对某单一栏来处理。我们再以表格的范例数据，对下面的命令而言，只有性别(Sex)和年龄(Age)皆相同者，才会被视为重复的数据记录：

```
SELECT DISTINCT Sex, Age FROM employee
```

传回结果如下：

M	F	M	M	F
25	24	31	33	31

我们可以发现只剩下五笔数据，因为其中性别(Sex) = M、年龄(Age) = 25 有两笔相同数据，因此只显示出一笔。

13-4-3 查询结果之输出目的地

一般情况，SELECT 的查询结果会传回给前端应用程序做处理，但是有时希望将查询结果输出至另一储存处，以便进行更进一步的处理。此时我们利用 INTO 自变量，便可以新建一个数据表并将查询结果存入其中。以下笔者举个范例来说明如何使用 INTO 自变量，将查询结果储存至数据库中的新数据表 MyTempTable 中：

```
SELECT * INTO MyTempTable FROM employee
```

欲在 SELECT 命令中加入 INTO 自变量来将查询结果存放至一个新数据表中，有下列三点注意事项：

- SELECT 命令语句中不能加入 COMPUTE 自变量；
- SELECT 命令语句不能位于一个交易(Transaction)中；
- 您必须拥有目的数据库的 CREATE TABLE 权限。

13-4-4 WHERE 自变量的使用

WHERE 自变量的功能就是设定查询数据所须符合的条件。当针对单一数据表进行查询处理时，除非要过滤数据记录，否则根本不须使用 WHERE 自变量。在 WHERE 自变量的过滤条件式中，SELECT 可以利用表 13-14 所示的运算符进行比对的处理。

表 13-14

运算符	说 明
=	等于
>	大于
<	小于
<> 或 !=	不等于
<=	小于或等于
>=	大于或等于
!>	不大于
!<	不小于
IN	判断数据是否在指定的各个数据值中
BETWEEN	判断数据是否在指定范围的数据值中
LIKE	判断数据是否符合指定的格式

SELECT 命令中之过滤条件式的数目并没有限制，您也可以视需要来使用逻辑运算符 AND、OR 或 NOT。接下来笔者举几个简单的例子：

假若 employee 数据表中有六名员工，他们的性别(Sex)和年龄(Age)分别为：

员工编号(Number)	1	2	3	4	5	6
性别(Sex)	M	F	M	M	M	F
年龄(Age)	25	24	25	31	33	31

1. 查询 employee 数据表中男性的年龄分布情况

```
SELECT DISTINCT Age FROM employee WHERE Sex IN ('M ')
```

➔ 回传 25, 31, 33

2. 查询 employee 数据表中年龄在 25 ~ 32 之间的员工编号

```
SELECT Number FROM employee WHERE Age BETWEEN 25 AND 32
```

➔ 回传 1, 3, 4, 6

或者也可以用：

```
SELECT Number FROM employee WHERE Age >= 25 AND Age <= 32
```

➔ 回传 1, 3, 4, 6

上述两种写法代表相同的意思，因此可以推知，第一种写法中的 BETWEEN 25 AND 32 是包含 25 和 32。

13-4-5 汇总函数

SELECT 命令的数据域位 select_expression、GROUP BY 自变量、HAVING 自变量中可以使用汇总函数，以便迅速取得数据记录的相关统计信息。一般而言，常用到的汇总函数主要有五个，它们分别为：COUNT()、MAX()、MIN()、AVG()和SUM()。接下来笔者将依序介绍这五个常用的汇总函数：

● COUNT({ [ALL | DISTINCT] expression } | *)

汇总函数 COUNT() 主要用来计算查询结果的数据笔数。我们通常以万用字符 “*” 作为 COUNT() 的自变量。注意一点，COUNT() 函数是惟一允许使用万用字符作为自变量的汇总函数。接下来看几个范例：

假若 employee 数据表中，有六名员工，他们的性别(Sex)和年龄(Age)分别为：

员工编号(Number)	1	2	3	4	5	6
性别(Sex)	M	F	M	M	M	F
年龄(Age)	26	27	26	31	33	31

1. 计算出 employee 数据表总共有几笔数据

```
SELECT COUNT(*) FROM employee
```

→ 回传 6

2. 计算出 employee 数据表中，男性员工有几人

```
SELECT COUNT(*) FROM employee WHERE Sex = 'M'
```

→ 回传 4

● MAX(expression | column_name)

汇总函数 MAX() 主要用于计算出一指定字段的最大值。我们一样用上面的表格为数据来当做下面的范例：

1. 计算出 employee 数据表中，年龄最大为几岁

```
SELECT MAX (Age) FROM employee
```

→ 回传 33

2. 计算出 employee 数据表中，女性年龄最大为几岁

```
SELECT MAX (Age) FROM employee WHERE Sex = 'F'
```

→ 回传 31

● MIN(expression | column_name)

汇总函数 MIN() 主要用于计算出一指定字段的最小值。这和 MAX() 函数的功能相近，只是一个找最大值，另一个找最小值。

● AVG(expression | column_name)

汇总函数 AVG() 主要用于计算出一指定字段的平均值。我们一样用上面的表格为数据来当做下面的范例：

1. 计算出 employee 数据表中，平均年龄为几岁

```
SELECT AVG (Age) FROM employee
```

→ 回传 29

2. 计算出 employee 数据表中，女性平均年龄为几岁

```
SELECT AVG (Age) FROM employee WHERE Sex = 'F'
```

→ 回传 29

● SUM(expression | column_name)

汇总函数 SUM() 主要用于计算出一指定字段的总和值。这和 AVG() 函数的功能相近，只是一个找平均值，另一个总和值。

13-4-6 SELECT 的万用字符

SELECT 命令语句中，有五个万用字符可以使用，它们分别为：*(星号)、%(百分比符号)、_(下画线符号)、[] (一对中括号) 和 [^]。其中除了对 *(星号) 做过介绍外，其他均未介绍过，接下来笔者针对这五个万用字符做个说明：

■ *(星号)

*(星号) 能使用在 select_expression 自变量中和 COUNT() 函数之中，其中 * 代表数据表中所有的字段。以下的命令语句，表示查询出 employee 数据表所有字段的内容：

```
SELECT * FROM employee
```

■ %(百分比符号)

%(百分比符号) 只能使用在 WHERE 自变量中，% 代表 0 个或 0 个以上的字符。比方说，ABC% 表示开头为 ABC 的字符串。%(百分比符号) 通常与运算符 LIKE 搭配使用。以下笔者举几个例子，以便读者能更了解如何使用 %(百分比符号) 万用字符。

1. 以下的命令语句能够查询出 employee 数据表的员工姓名中包含 '林' 者

```
SELECT * FROM employee WHERE Name LIKE '%林%'
```

2. 以下的命令语句能够查询出 employee 数据表中，姓 '王' 的员工姓名和年龄

```
SELECT Name, Age FROM employee WHERE Name LIKE '王%'
```

上面两个范例写法有点类似，但是代表的意思却截然不同。第一个范例中，Name LIKE '%林%' 表示只要员工姓名中包含 '林' 者都为符合条件。例如：林上杰、陈林中、令狐小林等等。

第二个范例中, `Name LIKE '王%'` 表示只有姓'王'的员工才符合条件。例如: 王大明、王小中、王小林等等。其实两者的差异在于 `%` 的位置, 前者是左右皆有 `%`, 如: `%林%`、后者只有右边有 `%`, `王%`, 因此希望读者务必要小心使用。

■ `_`(下画线符号)

下画线符号 `_` 代表单个字符, 它只能使用在 `WHERE` 自变量中。比方说, `_A` 表示第二个字符为 `A` 的字符串。下画线符号 `_` 通常与运算符 `LIKE` 搭配使用。以下笔者举几个例子。

1. 以下的命令语句能够查询出 `employee` 数据表之员工姓名为 'xx 珊'

```
SELECT * FROM employee WHERE Name LIKE ' _ 珊 '
```

2. 以下的命令语句能够查询出 `employee` 数据表中, 员工姓名第三个字为 '珊'

```
SELECT * FROM employee WHERE Name ' _ 珊 % '
```

■ `[]`(中括号)

中括号 `[]` 只能使用在 `WHERE` 自变量中, `[]` 用来限定任何一个单一字符介于指定的范围或集合中。一对中括号 `[]` 通常与运算符 `LIKE` 搭配使用。从以下的几个例子, 读者应该更能了解一对中括号 `[]` 万用字符。

1. 下面的查询命令语句中的过滤条件 `WHERE Name LIKE '[P-Z]inger'`, 表示找出名字的第一个字符为字母 `P~Z` 之间且后五个字符为 `inger` 者, 例如: `Ringer`、`Singer`、`Uinger` 等等, 皆为符合条件者:

```
SELECT * FROM employee WHERE Name LIKE '[P-Z]inger'
```

2. 下面的查询命令语句中的过滤条件 `WHERE Name LIKE '[ACD 林王]%'`, 表示找出名字的第一个字符为 `A`、`C`、`D`、`林`或`王`者, 例如: `Alex`、`Cindy`、`Doggie`、`林上杰`、`王小明`等等, 皆为符合条件者:

```
SELECT * FROM employee WHERE Name LIKE '[ACD 林王] % '
```

3. 下面的查询命令语句能够查询出姓'林'且第二个字为 '上' 或 '小' 的姓名, 例如: `林上杰`、`林小明`、`林上文子`等等, 皆为符合条件者:

```
SELECT Name FROM employee WHERE Name LIKE '林[上小] % '
```

■ `[^]`

`[^]` 只能使用在 `WHERE` 自变量中, `[^]` 用来限定任何一个单一字符不介于指定的范围或

集合中。[^]和[]的差别在于[^]是不介于指定的范围或集合中；而[]是介于指定的范围或集合中。笔者还是举几个例子，方便读者能理解。

(1) 查询命令语句中的过滤条件 `WHERE Name LIKE '[^H-K]ichel'`，表示找出名字的第一个字符不介于字母 H~K 之间(亦即必须介于 A~G 或 L~Z)且后五个字符为 inger 者，例如：Michel、Lichel、Aichel 等等，皆为符合条件者：

```
SELECT * FROM employee WHERE Name LIKE '[^H-K]ichel'
```

(2) 下面的查询命令语句中的过滤条件 `WHERE Name LIKE '[^ACD 陈许]%'`，表示找出名字的第一个字符不为 A、C、D、陈或许者，例如：John、Mike、Marry、林上杰、王小珊等等，皆为符合条件者：

```
SELECT * FROM employee WHERE Name LIKE '[^ACD 陈许]%'
```

(3) 下面的查询命令语句能够查询出姓 '林' 且第二个字不为 '下' 或 '大' 的姓名，例如：林上杰、林小明、林上文子等等，皆为符合条件者：

```
SELECT Name FROM employee WHERE Name LIKE '林[下大]%'
```

以上简单介绍完五个万用字符之后，接下来要谈论一下使用万用字符会遇到的问题。一般而言，如果按照上述的说明来操作万用字符时，应该不会出现什么问题，不过，现在有一个状况是读者可能会遇到的，那就是当读者的文字数据中正好就包含星号 *、百分比符号 % 或是下划线符号 _ (例如：mike_lin)，此时该如何解决？

当遇到此种情况时，读者可以利用 `ESCAPE` 自变量告知 SQL Server 哪一个字符是属于数据而非万用字符。

查询姓名中包含下划线符号_者

```
SELECT Name FROM employee WHERE Name LIKE '%\_%' ESCAPE '\'
```

此时位于 `ESCAPE` 自变量所指定字符(即 \)后的那一个字符(即 _)将被视为常数字符数据值而不是万用字符。以专业术语来说，`ESCAPE` 自变量所指定的字符称之为转义字符，因此上面范例的 \ 即为转义字符。

13-4-7 数据分组小计——GROUP BY

前面已经提过如何利用汇总函数来获取查询结果中某一直行“所有”数据的统计值，但是或许读者想更进一步计算出各个分组数据的统计值时，就必须借助 `GROUP BY` 的语句，如下所示：

```
GROUP BY [ALL] group_by_expression [,...n]
```

利用 `GROUP BY` 自变量，我们可以根据一个或一个以上直行的值来将查询中的数据记录分组。语句中的 `group_by_expression` 可以是一个数据表字段或是一个由字段所组成且不包含

汇总函数的表达式。接下来笔者举个数据分组的范例来说明如何使用 GROUP BY 自变量。

假若 employee 数据表中有六名员工，他们的性别(Sex)和年龄(Age)分别为：

员工编号(Number)	1	2	3	4	5	6
性别(Sex)	M	F	M	M	M	F
年龄(Age)	26	27	26	31	33	31

计算出 employee 数据表中，男、女各别年龄最大值和最小值：

```
SELECT Sex AS 性别,
       MAX(Age) AS 年龄最大值,
       MIN(Age) AS 年龄最小值
FROM employee
GROUP BY Sex
```

结果如下：

性别	年龄最大值	年龄最小值
M	33	26
F	31	27

13-4-8 HAVING 自变量的使用

HAVING 自变量主要是和 GROUP BY 自变量合用的，以便用来过滤分组数据。HAVING 自变量可以包含多个过滤条件式，而这些过滤条件式之间是利用 AND 或 OR 运算符相连接的，当然也可以利用 NOT 运算符来逆转一个布尔表达式。

或许读者发现到 HAVING 自变量和 WHERE 自变量的相似之处，HAVING 自变量主要用来过滤分组后数据；WHERE 自变量则用来过滤个别的数据。假若当 HAVING 自变量未和 GROUP BY 自变量搭配使用时，HAVING 自变量的作用则和 WHERE 自变量类似，不过笔者还是建议读者要过滤个别的数据记录时，还是使用 WHERE 自变量而不要使用 HAVING 自变量。接下来笔者举个数据分组的范例来说明如何使用 HAVING 自变量，这里一样是使用 GROUP BY 的数据表格为数据模板。

(1) 计算出 employee 数据表中，年龄大于 26 之男、女各别平均年龄：

```
SELECT Sex AS 性别,
       AVG(Age) AS 平均年龄
FROM employee
GROUP BY Sex
HAVING Age > 26
```

结果如下：

性别	平均年龄
M	32
F	29

男性平均年龄为 32 的算法: $(31 + 33) / 2 = 32$

女性平均年龄为 29 的算法: $(27 + 31) / 2 = 29$

其中有两位男性(M)的年龄小于 26, 所以此两笔数据不予计算。

(2) 计算出 employee 数据表中, 哪个年龄是员工重复的年龄且重复个数为多少:

```
SELECT Age AS 相同年龄,
       COUNT(*) AS 相同年龄的人数
FROM employee
GROUP BY Age
HAVING COUNT(*) > 1
```

结果如下:

相同年龄	相同年龄的人数
26	2
31	2

13-4-9 ORDER BY 自变量的使用

利用 ORDER BY 自变量可以根据一个或一个以上字段之数据来排序查询结果, ORDER BY 的语句如下所示:

```
ORDER BY order_by_expression [ASC | DESC] [,... n]
```

语句中的 order_by_expression 可以是数据表中的一个字段、表达式或是查询结果中某一个字段或表达式的位置顺序编号。一般在默认状态下, SQL Server 会根据您所指定的 order_by_expression 依升幂的次序排序查询结果, 假如您希望依降幂的次序排序查询结果时, 可以在 order_by_expression 之后再加入关键字 DESC。接下来笔者举几个排序查询结果的范例来说明如何使用 ORDER BY 自变量。

假若 employee 数据表中, 有六名员工, 他们的性别(Sex)和年龄(Age)分别为:

员工编号(Number)	1	2	3	4	5	6
性别(Sex)	M	F	M	M	M	F
年龄(Age)	26	27	26	31	33	31

依照年龄的高低顺序, 列出 employee 数据表所有员工的编号、性别和年龄:

```
SELECT Number AS 员工编号,
       Sex AS 性别,
       Age AS 年龄
```

JSP2.0 技术手册

```
FROM employee  
ORDER BY Age DESC
```

结果如下：

员工编号	性别	年龄
5	M	33
4	M	31
6	F	31
2	F	27
1	M	26
3	M	26

上述的命令语句也可以改为下列所示：

```
SELECT Number AS 员工编号,  
       Sex AS 性别,  
       Age AS 年龄  
FROM employee  
ORDER BY 3 DESC
```

差别在于 ORDER BY Age DESC 改为 ORDER BY 3 DESC，因为 Age 是代表查询结果的第三个字段顺序，因此可以把 Age 改为 3。假若是 ORDER BY Sex DESC 时，亦可改为 ORDER BY 2 DESC。

13-4-10 SELECT 查询命令总结

笔者总共花九个小节来解说 SELECT 查询命令的使用方式，最后笔者再做一个总结。SELECT 查询命令语句如下所示：

```
SELECT [DISTINCT | ALL] select_expression  
      [INTO {OUTFILE | DUMPFILE} 'file_name' export_options]  
FROM table_references  
[WHERE where_definition]  
[GROUP BY col_name,...]  
[HAVING where_definition]  
[ORDER BY {unsigned_integer | col_name | formula} [ASC | DESC] ,...]
```

本小节笔者介绍 SELECT 查询命令的结构，就是依照上面 SELECT 语句结构的顺序来做说明介绍的，如表 13-15 所示。

除此之外，笔者在 13-4-5 和 13-4-6 中介绍了汇总函数和万用字符的使用，以补齐 SELECT 查询命令的完整性，希望经过一连串的说明之后，读者能够学会使用 SELECT 命令来查询数据。接下来的章节将会为各位读者介绍其他 DML 语言（Data Manipulation Language）的成员，它们包括：INSERT、UPDATE 和 DELETE。

表 13-15

语 句	说 明
[DISTINCT ALL] select_expression	13-4-1 和 13-4-2 小节主要分别介绍 select_expression 和 DISTINCT ALL 的用法
[INTO {OUTFILE DUMPFILE} 'file_name' export_options]	13-4-3 小节介绍使用 INTO、OUTFILE 自变量来将查询结果输出至目的地
[WHERE where_definition]	13-4-4 小节介绍 WHERE 自变量的使用
[GROUP BY col_name,...]	13-4-7 小节介绍 GROUP BY 自变量的使用
[HAVING where_definition]	13-4-8 小节介绍 HAVING 自变量的使用
[ORDER BY {unsigned_integer col_name formula} [ASC DESC] ,...]	13-4-9 小节介绍 ORDER BY 自变量的使用

13-5 新增数据——INSERT

一般而言，SQL Server 都有两个命令语句用来新增数据记录，它们的语句分别为：

第一种：

```
INSERT INTO [ owner_name.] table_name
      [ ( column1 [,column2 [, ...]] ) ]
VALUES ( expression1 | NULL | DEFAULT
      [,expression2 | NULL | DEFAULT , ..... ] ) | DEFAULT VALUE
```

第二种：

```
INSERT INTO [ owner_name.] table_name
      [ ( column1 [,column2 [, ...]] ) ]
SELECT select_statement
```

接下来就分别说明这两种新增数据命令的用法，并且搭配几个范例，让读者能够清楚且快速了解它们的使用方法。

13-5-1 INSERT INTO ... VALUES

我们可以使用 INSERT 命令将一笔数据记录新增至 table_name 所指定的数据表；VALUE 自变量中的 expression1, expression2... 是用来指定所要新增的数据值，至于这些数据值要被新增至哪一个字段中，则是由 column1, column2 ...所指定的字段。expression1 自变量的数据值会被加至 column1 自变量所指定的字段，expression2 自变量的数据值会被加至 column2 自变量所指定的字段，其余依此类推。但是要注意一点，如果您并未指定字段名称，则 expression1, expression2 等数据值的次序会和数据表中各个字段的次序相同。

在新增数据的过程中，您所新增数据的类型要和字段的类型相一致，否则会无法顺利将数据新增至数据表中，当然也不能违反数据表的各项条件约束，例如：NOT NULL 等等。下面的范例中，我们先新建一个数据表 MyTable，然后使用 INSERT 命令将数据记录新增至数据表中：

• 建立数据表

```
CREATE TABLE MyTable
(
    Name varchar(20),
    Sex bit,
    Birthday datetime,
    Tel varchar(16)
)
```

• 新增数据记录

```
INSERT INTO MyTable ( Name, Sex, Birthday, Tel )
VALUES ( 'Mike', 1, 12/27/1978, '2301-5896' )

INSERT INTO MyTable ( Name, Sex, Birthday, Tel )
VALUES ( 'Shan', 0, 08/04/1982, '2304-5096' )
```

13-5-2 关键字 DEFAULT

从 INSERT 命令语句中可以看出，我们能够以关键字 DEFAULT 作为要新增至字段中的数据值。此时 SQL Server 将依下列顺序来决定要将什么数据值新增至对应的字段中：

- 如果字段有默认值时，SQL Server 会将其默认值新增至字段中；
- 如果字段没有默认值且此字段允许 NULL 值时，SQL Server 会将 NULL 值新增至字段中；
- 如果字段没有默认值且此字段不允许 NULL 值时，INSERT 命令将会失败。

下面的范例中，我们先新建一个数据表 MyTable，并且设定 Sex 默认值为 1。

• 建立数据表

```
CREATE TABLE MyTable
(
    Name varchar(20) NOT NULL,
    Sex bit DEFAULT 1,
    Birthday datetime NULL,
    Tel varchar(16) NULL
)
```

• 新增数据记录

```
INSERT INTO MyTable ( Name, Sex, Birthday, Tel )
VALUES ( 'Mike', DEFAULT, 12/27/1978, '2301-5896' )

INSERT INTO MyTable ( Name, Sex, Birthday, Tel )
VALUES ( 'Shan', 0, DEFAULT, '2304-5096' )
```

结果如下：

Name	Sex	Birthday	Tel
Mike	1	12/27/1978	2301-5896
Shan	0	NULL	2304-5096

其实有一个方法, 可以不用 VALUES 自变量来一一指定要新增至字段中的数据值, 而直接加入关键字 DEFAULT VALUES 即可。例如:

```
INSERT MyTable DEFAULT VALUES
```

如果您使用这种新增命令时, SQL Server 会依下列原则来新增数据至各个字段中:

- 如果字段有默认值时, SQL Server 会将其默认值新增至字段中;
- 如果字段没有默认值且此字段允许 NULL 值时, SQL Server 会将 NULL 值新增至字段中;
- 如果字段没有默认值且此字段不允许 NULL 值时, INSERT 命令将会失败。

由上面的说明可以了解, 当您以加入关键字 DEFAULT VALUES 的方式来新增数据记录时, 只要有一个字段没有默认值此字段不允许 NULL 值时, INSERT 命令将会失败。因此在正常的数据新增作业之中不可能会使用到关键字 DEFAULT VALUES, 它主要的用途在于产生一些样本数据。

13-5-3 INSERT INTO ... SELECT

新增数据记录的第二种方法, 就是使用 INSERT INTO ... SELECT 命令, 其语句如下所示:

```
INSERT INTO [ owner_name.] table_name
           [ ( column1 [,column2 [, ...]] ) ]
SELECT select_statement
```

INSERT INTO ... SELECT 命令最大的特色在于: 它能够将查询结果直接储存至一个数据表中, 通过这种做法, 我们能够很轻易将一个或一个以上之数据表中的数据记录经过汇总、分析、统计和运算后, 再存放至某一数据表中。笔者举个范例程序, 如下所示:

- 建立数据表

```
CREATE TABLE MyTable
(
    Name varchar(20) NOT NULL,
    Sex bit DEFAULT 1,
    Birthday datetime NULL,
    Tel varchar(16) NULL
)
```

- 新增数据记录

```
INSERT INTO MyTable ( Name, Sex, Birthday, Tel )
SELECT ( Name, Sex, Birthday, Tel ) FROM HisTable
WHERE Sex = 1
```

上面程序代码所代表的意思是: 查询 HisTable 数据表中, Sex = 1(即男性)的所有数据, 并且将它储存至 MyTable 数据表。

JSP2.0 技术手册

13-6 修改数据——UPDATE

假若我们要修改数据表中的数据时，我们必须使用 UPDATE 命令，它的语句如下：

```
UPDATE table_name
    SET ( column1 = { expression | DEFAULT | NULL } ) [,column2 ...]
    WHERE search_conditions
```

UPDATE 命令语句中的 SET 自变量用来指定将要被更新的字段与用来替代旧值的新值，其中 column1 自变量是指定将要被更新之字段名称；而 expression 自变量则是指定要用来替代旧数据值的新数据值，expression 可以是一个常数、变量、表达式或是一个传回单一值的子查询。

如果您希望将字段中的数据值更改成此字段的默认值时，可以使用关键字 DEFAULT，但如果字段未定义默认值且该字段允许 NULL 值时，则字段内容将被更改成 NULL 值；另外假若您希望将字段中的数据值更改成 NULL 值时，可以使用关键字 NULL。下面笔者将举几个例子来说明 UPDATE 命令的使用方式。

- (1) 将数据表 MyTable 中的 Name 字段所有数据记录全部更改为 'Mike'：

```
UPDATE MyTable SET Name = 'Mike'
```

- (2) 将数据表 MyTable 中的 Salary 字段大于 8000 元者皆减薪 500 元：

```
UPDATE MyTable SET Salary = Salary - 500 WHERE Salary > 8000
```

13-7 删除数据——DELETE

假若我们要删除数据表中的数据时，我们必须使用 DELETE 命令，它的语句如下：

```
DELETE FROM table_name
    WHERE search_conditions
```

DELETE 命令用来删除 table_name 自变量所指定数据表中的数据记录，而 WHERE 自变量用来设定符合特定条件的数据记录才会被删除，这一过滤条件即为语句中的 search_conditions。下面笔者将举几个例子来说明 DELETE 命令的使用方式。

- (1) 将数据表 MyTable 中所有数据记录全部删除：

```
DELETE FROM MyTable
```

- (2) 将数据表 MyTable 中的 Salary 字段大于 8000 元者删除：

```
DELETE FROM MyTable WHERE Salary > 8000
```


14

第十四章

JSP 与 JDBC

本章主要探讨 JDBC 的架构与运作方式，让读者对 JDBC 有完整的基础概念后，能够快速上手 JSP 与数据库的运用。至于 JDBC API 本身以及详细的内容，建议读者参考其他相关 JDBC 的书籍，以便得到更详细的说明。本章分 10 节来介绍：

- 14-1 JDBC 简介
- 14-2 MySQL 的安装与使用
- 14-3 JDBC 连接 MySQL
- 14-4 JDBC 连接 MySQL 的中文问题
- 14-5 PreparedStatement
- 14-6 CallableStatement
- 14-7 JDBC 2.0 介绍与使用
- 14-8 JNDI –数据来源(Data Source)与连接池(Connection Pool)
- 14-9 JSTL 的 SQL 标签库
- 14-10 Connection Pool – Proxool

JSP2.0 技术手册

14-1 JDBC 简介

JDBC 是一种用于执行 SQL 的 Java API (JDBC 本身是个商标名,但是 JDBC 常被误认为是“Java 数据库连接(Java Database Connectivity)”的缩写。JDBC 为数据库开发人员提供了一个标准的 API,使用纯 Java API 来开发数据库应用程序。

升阳和主流数据库厂商合作开发出一组用于存取数据库的 JDBC API。在研发过程中,追求三个目标:

- (1) JDBC 应为 SQL 层次的 API;
- (2) JDBC 应借重现在数据库 API 的经验;
- (3) JDBC 要简单方便使用。

所谓 SQL 层次的 API 意指:JDBC 可以让你建立 SQL 语句,且将它内嵌到 Java API 中。JDBC 让你在数据库与 Java 应用程序之间来去自如,例如:你从数据库中得到的结果,能被当成 Java 对象传回来,如果存取出了问题时,应用程序也会抛出异常事件。

由于专用的数据库存取 API 为数众多,造成许多的困扰,因而设计一通用、标准的数据库存取 API 来解决这个问题的想法并不是没有。事实上,升阳曾经公开借用 ODBC(Open DataBase Connectivity,开放式数据库连线作业)的优点,例如:ODBC 在 Windows 环境中为单一的数据库存取标准。虽然业界已认定 ODBC 是和 Windows 数据库对话的主要方式,但在 Java 世界中,ODBC 没有办法移转得很好。

首先,ODBC 是 C 写成的 API,对其他的语言来说,则需要中介的 API 来取得和 ODBC 做沟通联结,但是由于 ODBC 的设计过于复杂,即使是 C 语言的开发人员,要将 ODBC 转移到非 Windows 环境,也会面临到困难。ODBC API 的复杂起因于:某些复杂罕见的任务和其他较简单常用的机能混在一起,换言之,你为了多了解一点 ODBC,反而却要先学习更多的东西。

JDBC 尽可能维持操作的方便与简单,同时还能够提供研发人员最大的弹性。升阳采用的关键准则是:只在乎数据库存取应用程序时,开发人员是否能够轻松存取数据。简单常见的功能依循简单的接口设计(interface),而较罕见的任务则通过特殊接口来解决,例如:一般的数据库存取,通常由三个接口来做处理,不过 JDBC 也提供好几个接口用于处理较复杂罕见的任务。

14-2 MySQL 的安装与使用

上一节介绍了 JDBC 的概念之后,接下来将会偏向 JDBC 的实务操作。一般市场常见的数据库系统有:Oracle、MS SQL Server、IBM DB2、MySQL、PostgreSQL 和 Sybase 等等,其中市占率前三名的数据库系统为 Oracle、IBM DB2 和 MS SQL Server,它们无论在稳定性、安全性和性能上皆有不错的表现,因此广受市场的欢迎。

笔者选择 MySQL 作为本书的范例数据库,将用它来说明 JDBC 如何连接数据库、存取数据,进而对数据做分析或统计。或许有读者会问:为什么要选择 MySQL,而不选择功能更强大的 Oracle 或 IBM DB2?针对这个问题,笔者根据以下理由来说明:

JSP2.0 技术手册

- (1) MySQL 在一般用途上是免费的，而且容易取得，因此在学习上，我们不用担心版权问题。
- (2) MySQL 可以在多种平台下使用，目前 MySQL 支持的平台有：Linux、Windows、Solaris、FreeBSD、HP-UX、MacOS X、AIX 等等，因此我们也不用担心平台的问题。
- (3) MySQL 虽然是免费的数据库系统，但是它在性能表现上也有不错的成绩，因此广受大家的喜爱，尤其是 Linux 社群人士或是中小型私人公司。

14-2-1 安装 MySQL in WindowsXP

1. 读者可以自行至 MySQL 的官方网站 <http://www.mysql.com> 下载新版的 MySQL，或是直接使用光盘内附 MySQL 4.0.16，文件名为 *mysql-4.0.16-win.zip*。
2. 将 *mysql-4.0.16-win.zip* 解压缩之后，直接执行安装程序。MySQL 默认的安装路径为 *c:\mysql*。
3. 安装成功后，打开一个指令控制台(command console)，然后依序输入(见图 14-1)
`cd c:\mysql\bin`
(目录路径会移到 *c:\mysql\bin* 下)

`c:\mysql\bin>mysqld-nt --install`

这样一来，当您的计算机重新启动后，MySQL 会自动启动。

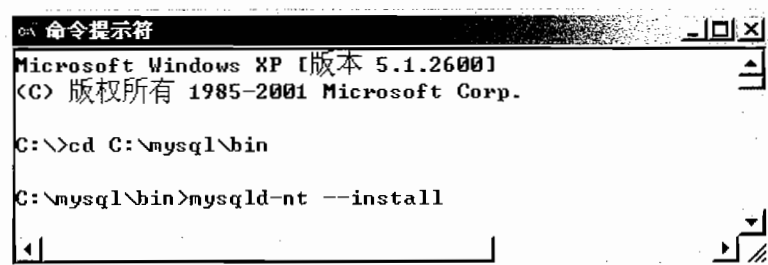


图 14-1 安装 MySQL

4. 再来执行 *C:\mysql\bin* 中的 *winmysqladmin*。*winmysqladmin* 工具可以让你控制 MySQL 服务器。一开始执行 *winmysqladmin* 时，会先要求你设定用户名称和密码，设定之后就能进入 *winmysqladmin*。
5. 测试 MySQL 是否有启动，至 *c:\mysql\bin* 目录下，输入下列指令：

`c:\mysql\bin>mysqlshow`

如果执行画面和图 14-2 一样，那表示您已经成功安装好 MySQL。

在图 14-2 中，我们发现 MySQL 安装成功后，MySQL 已经先默认建立两个数据库：*mysql* 和 *test*。

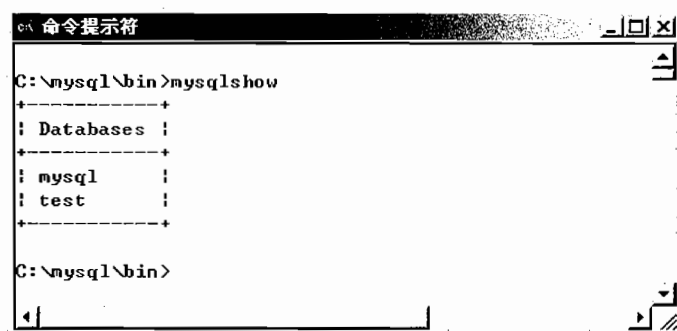


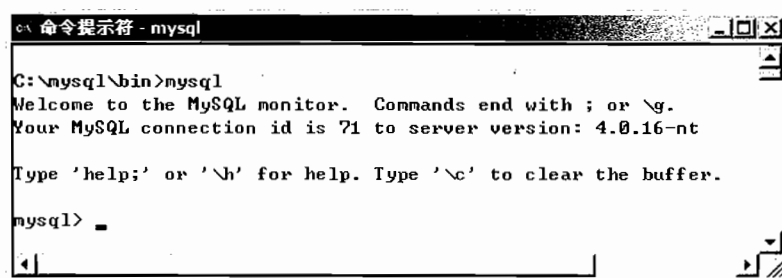
图 14-2 测试 MySQL 是否启动

14-2-2 MySQL 的基本安全设定

上一章节我们已经成功安装了 MySQL，接下来我们须用到 `c:\mysql\bin` 目录下的 `mysql` 指令工具，即在 `c:\mysql\bin` 目录下输入下列指令：

```
c:\mysql\bin >mysql
```

指令执行后，它会启动在其原来的窗口中编辑而成的 `mysql` 工具，如图 14-3 所示：

图 14-3 在 `c:\mysql\bin` 目录下执行 `mysql` 指令

MySQL 安装在 Windows 环境时，默认的 MySQL 权限是允许所有的用户都有完整存取数据库的权限，例如：能执行 `CREATE DATABASE`(建立数据库)、`DROP DATABASE`(删除数据库)等等。这样的问题是非常严重且不安全的，因此，为让 MySQL 能够更加安全，我们必须为 MySQL 的 `root` 设定密码，并且设定不允许所有的用户有完整存取数据库的权限。

首先我们先设定不允许所有的用户有完整存取数据库的权限。在 MySQL 中有一名称为 `mysql` 的数据库，`mysql` 数据库有六个数据表：`columns_priv`、`db`、`func`、`host`、`tables_priv` 和 `user`。其中 `user` 这个数据表定义用户的相关权限，因此，若要取消默认用户有完整存取数据库的权限，我们的做法只要如下：

```
mysql> USE mysql;  
mysql> DELETE FROM user WHERE User='_';  
mysql> QUIT
```

JSP2.0 技术手册

第一个指令代表我们选定使用 mysql 这个数据库。第二个指令代表删除 user 数据表中 User = '' 的数据。

接下来我们为 root 设定密码，做法如下：

```
c:\mysql\bin > mysqladmin reload
```

```
c:\mysql\bin > mysqladmin -u root password your_password
```

我们在这里使用另一个指令工具 mysqladmin 来设定 root 的密码，其中 your_password 即为读者自行设定的密码。

做完上述的动作，已经让 MySQL 更为安全了，假若您想要关掉 MySQL，那么您可以使用下列的指令：

```
c:\mysql\bin>mysqladmin --user=root --password= your_password shutdown
```

除了上述的指令之外，您也可以直接使用 net stop mysql 指令来关掉 MySQL；反之，若要重新启动 MySQL，可以直接使用 net start mysql。

```
c:\mysql\bin > net start mysql
```

现在我们就利用 root 的身份，联机进入 MySQL 服务器，指令如下：

```
c:\mysql\bin >mysql -h localhost -u root -p
```

按下【Enter】后，读者应该会看到：

Enter password: *****

上述的星号 (*****) 代表您设定的密码。成功登录之后，画面应该会和图 14-4 一样。

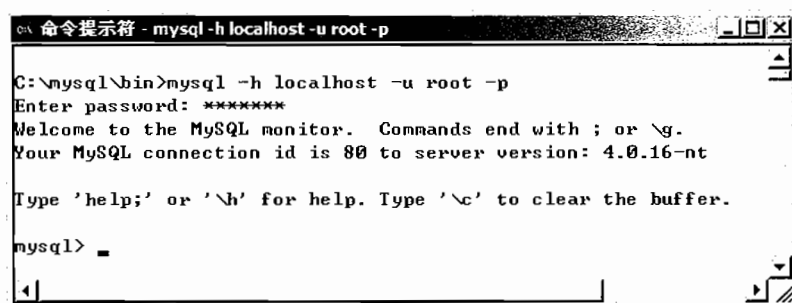


图 14-4 成功登录 root 身份

14-2-3 MySQL 常用的指令

MySQL 提供一些好用的指令来操作数据库，一般常见的指令如表 14-1。

大部分的指令都可从名称猜出指令的用途，例如：USE test 表示使用 test 数据库；SHOW DATABASES 表示列出所有的数据库，等等。假若我们要查看 mysql 数据库中有哪些数据表时，可以用如下的指令：

```
mysql> USE mysql;
```

```
mysql> SHOW TABLES;
```

表 14-1

指 令	说 明
USE db_name	选定将使用的数据库
SHOW DATABASES	列出 MySQL 中所有的数据库
SHOW TABLES	列出当前数据库中所有的数据表
SHOW TABLES FROM db_name	列出 db_name 中所有的数据表
SHOW COLUMNS FROM table_name	列出 table_name 完整信息，如：栏名、类型
SHOW INDEX FROM table_name	列出 table_name 中所有的索引
SHOW TABLE STATUS	列出当前数据库中数据表的信息
SHOW TABLE STATUS FROM db_name	列出 db_name 中数据表的信息
SHOW PROCESSLIST	列出每一笔联机的信息
SHOW VARIABLES	列出 MySQL 的系统设定
DESCRIBE table_name	列出 table_name 完整信息，如：栏名、类型
EXPLAIN table_name	列出 table_name 完整信息，如：栏名、类型

执行画面如图 14-5 所示。

```
命令提示符 - mysql -h localhost -u root -p
mysql> USE mysql;
Database changed
mysql> SHOW TABLES;
+-----+
| Tables_in_mysql |
+-----+
| columns_priv    |
| db              |
| func            |
| host            |
| tables_priv     |
| user            |
+-----+
6 rows in set (0.00 sec)

mysql>
```

图 14-5 查看 mysql 数据库中有哪些数据表

假若我们要查询 host 数据表中的字段信息时，可以使用如下的指令：

```
mysql> DESCRIBE host;
```

执行画面如图 14-6 所示。

除了 DESCRIBE table_name 可以查询数据表中的字段信息之外，EXPLAIN table_name 和 SHOW COLUMNS FROM table_name 一样也可以查询数据表中的字段信息，且它们三个指令的功能完全是一样的。

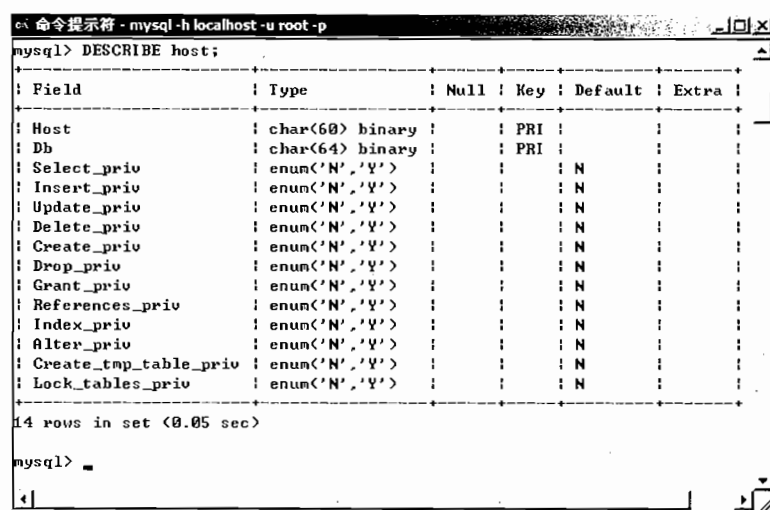


图 14-6 查询 host 数据表中的字段信息

14-2-4 MySQL 的 SQL 语句

本章要介绍 MySQL 中 DDL 语句的使用, 假若读者忘了什么是 DDL 语句时, 可以参阅 13-3 的内容。本章把 DDL 语句的使用分为两部分, 如表 14-2 所示:

表 14-2

与数据库有关的 DDL 语句	
CREATE DATABASE db_name	建立数据库
DROP DATABASE db_name	删除数据库
USE db_name	选定数据库
与数据表有关的 DDL 语句	
CREATE TABLE	建立数据表
ALTER TABLE	修改数据表
DROP TABLE	删除数据表
CREATE INDEX	建立索引
DROP INDEX	删除索引

与数据库有关的 DDL 语句

MySQL 提供三个与数据库有关的语句, 它们分别是: CREATE DATABASE 用于建立数据库, DROP DATABASE 用于删除数据库, USE 用于选择要使用数据库。

1. CREATE DATABASE

建立一个数据库很容易，只要在CREATE DATABASE语句中给出其名称即可：

```
CREATE DATABASE db_name
```

其中限制条件是：该数据库的名称必须是合法的、惟一的，并且您必须有足够的权限来建立它，例如：使用root的身份联机登录，才拥有足够的权限执行CREATE DATABASE。

2. DROP DATABASE

删除数据库就像建立它一样容易，假如有足够的权限，执行下列语句即可：

```
DROP DATABASE db_name
```

请注意，要小心使用DROP DATABASE语句，它将会删除数据库及其所有的数据表。

3. USE

USE 语句选择一个数据库，使其成为连接MySQL时的默认数据库：

```
USE db_name
```

必须拥有执行此数据库的权限，否则不能使用它。假若要使用某数据库的数据表时，可以直接利用db_name.table_name的形式来引用它的数据表。但是，如果都是使用同一数据库的数据表时，还是先设定联机的数据库，否则在引用数据表时，都要加上数据库限定词(db_name)，这样子反而麻烦。

选择一个默认数据库之后，我们还可以任意使用USE语句，在数据库之间进行任意地切换，只要具有使用它们的权限即可。选定好数据库也不限制您只能使用该数据库中的数据表，您仍然可以用db_name.table_name的方法，引用其他数据库的数据表。

接下来我们建立一个数据库sample_db，然后选定它为默认数据库。图14-7列出了目前MySQL所有的数据库：

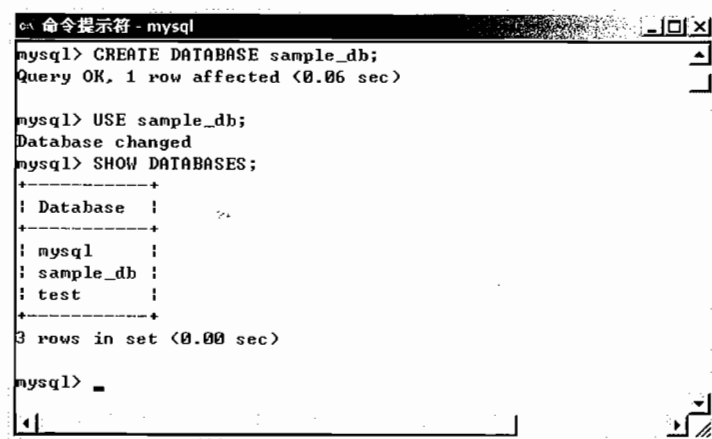


图 14-7 新增 sample_db 后，目前 MySQL 所有的数据库

■ 与数据表有关的 DDL 语句

我们可以利用 CREATE TABLE、DROP TABLE 和 ALTER TABLE 语句来建立数据表，对它们进行删除或更改它们的结构。不过 CREATE TABLE 和 ALTER TABLE 语句比较复杂，读者可以翻至 13-3-2，此章节针对这部分有详细的说明。本章节重心放在 MySQL 的实现上，因此笔者直接用范例来实务操作。CREATE INDEX 和 DROP INDEX 语句让您能够增加或删除现有数据表上的索引。

现在我们在 sample_db 数据库中新增一个数据表 employee，语法如下：

```
mysql> CREATE TABLE employee (
    employee_id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY ,
    last_name VARCHAR(15) NOT NULL ,
    first_name VARCHAR(15) NOT NULL ,
    birth DATE NOT NULL ,
    sex ENUM(' M', ' F' ) DEFAULT ' M' ) ;
```

我们建立 employee 数据表，它有五个字段，分别为：employee_id、last_name、first_name、birth 和 sex。

employee_id 代表员工的 ID 号码，它的定义比较复杂，它包括下列几个部分：

- INT：说明此行的值必须为整数，即无小数。
- UNSIGNED：不允许负数。
- NOT NULL：表示此行的值必须填入，即每个员工都有 ID 号码。
- AUTO_INCREMENT：是 MySQL 中一个特殊的属性。其作用为：如果在建立一笔新的 employee 数据记录时，遗漏了 employee_id 的值（或为 NULL），MySQL 会自动生成一个大于当前此行中最大值的惟一 ID 号。在新增员工数据表时将会用到这个特性，新增员工数据表时，可以填入 last_name、first_name、birth 和 sex 的值，让 MySQL 自动生成 employee_id 值。
- PRIMARY KEY：表示此行的值为快速搜索进行索引，并且行中的每个值都必须是惟一的。这样可以防止同一名字的 ID 出现两次，这对于员工 ID 号来说是一个必须的特性。

last_name 代表是姓氏，而 first_name 是代表名字。它们都是一个可变长度的字符串类型，最多存放 15 个字符，而且它们都不能为 NULL，即都必须填入适当的数据值。

birth 代表出生日期。它的类型为日期，格式：YYYY-MM-DD。

sex 表示员工的性别，默认值为 M(男)。它是一个 ENUM(列举)类型，表示只能明确地取列出的值，这里列出的值为 M 和 F，分别表示男和女。在某行只具有一组有限值时，ENUM 类型非常有用。我们也可以用 CHAR(1) 来代替它，但是 ENUM 可以明确规定为何值。顺便说一下，ENUM 行中的值不一定只能为单个字符，例如：此行还可以定义为 ENUM('male', 'female')。

建立完成后，我们利用 DESCRIBE 指令来查看 employee 数据表的内容，如图 14-8 所示：

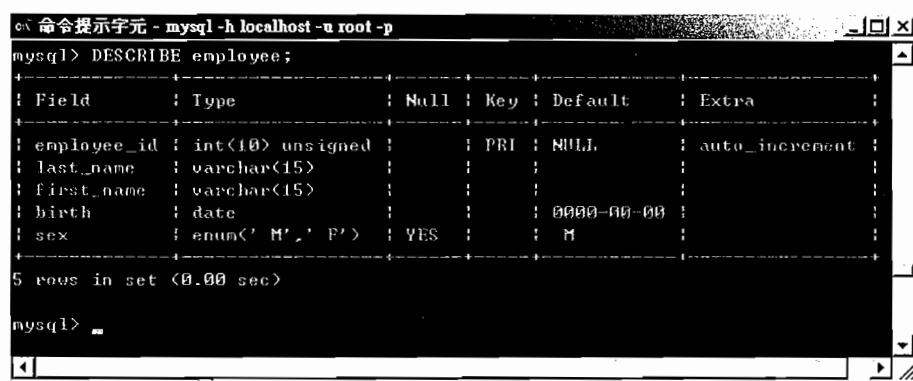


图 14-8 查看 employee 数据表的内容

当我们建立好数据表后，可以使用 ALTER TABLE 语句来修改数据表。ALTER TABLE 语句可以新增、变更、重新命名或移除现有数据表的字段。接下来我们看以下几个范例。首先我们在 employee 数据表再新增一个 email 字段，并且定义好此字段的属性：

```
mysql> ALTER TABLE employee ADD (email VARCHAR(40) NULL);
```

我们已经新增一个 email 字段，它是一个可变长度的字符串类型，最长可为 40 个字符，并且可以为 NULL。但是假若我们想要改变心意，想要加长 email 的可变字符串长度至 50，并且不允许 NULL 值时，我们可以这样做：

```
mysql> ALTER TABLE employee MODIFY email VARCHAR(50) NOT NULL;
```

如果我们要把 email 字段名称改为 e-mail 时，我们可以用下列语句：

```
mysql> ALTER TABLE employee CHANGE email e_mail VARCHAR(50) NOT NULL;
```

我们可以发现上述两个语句很像，一个是 MODIFY，另一个是 CHANGE。其实两者都可以用来变更字段的属性，但是 CHANGE 多了一样功能，即可以变更字段的名称，因此，笔者建议大家记住 CHANGE 的用法即可。

假若如果又不想要 e_mail 字段时，我们可以直接把它删除：

```
mysql> ALTER TABLE employee DROP e_mail ;
```

最后笔者还是保留 email 字段，因为之后的范例都会用到 employee 数据表的 email 字段，因此笔者又执行下面的语句：

```
mysql> ALTER TABLE employee ADD (email VARCHAR(40) NULL);
```

到此为止，employee 数据表总共有六个字段，分别为：employee_id、last_name、first_name、birth、sex 和 email。

14-3 JDBC 连接 MySQL

当我们使用 JDBC 后,就不必为连接 MySQL 数据库时专门写一个程序;连接 Oracle 数据库时,又要写另一个程序等等。读者只须利用 JDBC API 来实现开发程序,它就可以向任何数据库发送 SQL 语句执行工作。图 14-9 就是整个 JDBC API 与数据库之间的关系。

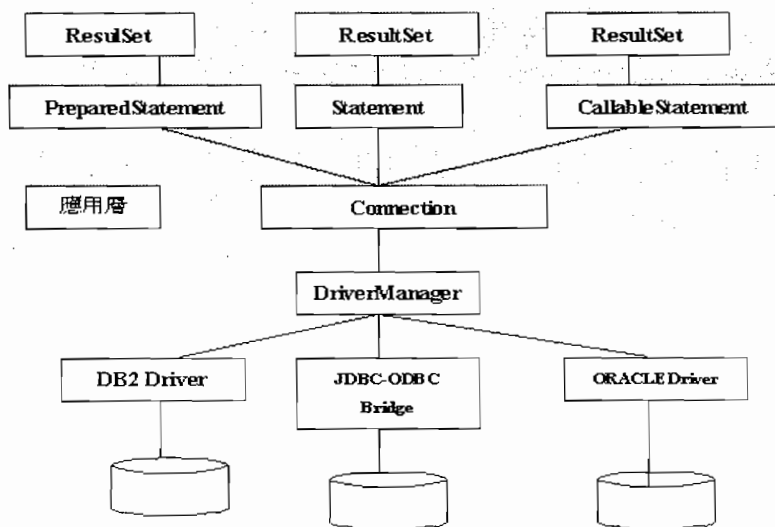


图 14-9 JDBC API 与数据库之间的关系

14-3-1 JDBC 驱动程序类型

我们仔细观察图 14-9,读者可以发现 JDBC API 分为两个不同的层次:应用层和驱动程序层。

■ 应用层

应用层次的 API,它主要是给数据库的应用程序开发者所使用的,换句话说:如果你只须要使用 SQL 语法来操作数据,如:新增、修改、删除或查询,那么了解应用层的 API 就足够了。

■ 驱动程序层

驱动程序的层次,则是撰写驱动程序的厂商,才须要用到的 API。因此当你要利用 JDBC 的 API 来开发你的程序时(这是应用层),你还要依赖厂商所提供的 JDBC 驱动程序,才能顺利

JSP2.0 技术手册

地与数据库连接。一般而言，厂商所提供的 JDBC 驱动程序可分为四大类：

TYPE 1: JDBC-ODBC 桥接驱动程序(JDBC-ODBC bridge)

TYPE 2: 原生 API 驱动程序(Native-API Driver)

TYPE 3: JDBC 通过网络的纯 Java 驱动程序

TYPE 4: 原生协议以及纯 Java 驱动程序

接下来就为各位介绍这四种类型的驱动程序：

● TYPE 1: JDBC-ODBC 桥接驱动程序(JDBC-ODBC bridge)

Type 1 的驱动程序提供将 JDBC 数据转换成 ODBC 数据来源，再利用 ODBC 来与数据库沟通。Sun 在 JDK 中附有免费的 JDBC-ODBC 桥接驱动程序，提供存取标准 ODBC 数据来源，如：Microsoft Access 之类的。不过 Sun 强烈建议 JDBC-ODBC 桥接器最好只是用来测试，不建议真正应用在我们的 Web 应用程序之中，主要原因在于：性能低落。图 14-10 说明了 JDBC-ODBC 桥接器概观。

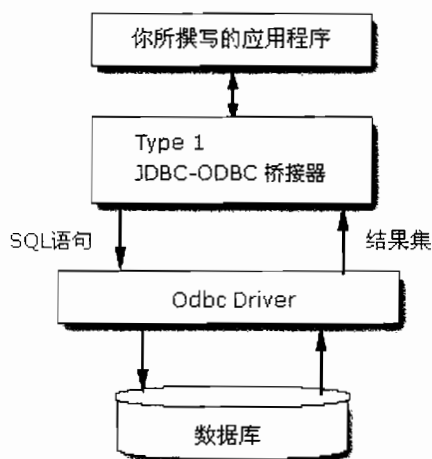


图 14-10 TYPE1 JDBC-ODBC 桥接器概观

● TYPE 2: 原生 API 驱动程序(Native-API Driver)

这个原生驱动程序是将 JDBC 指令转换成 DBMS 所指定的原生码。也就是说，有一套不是以 Java 所撰写的函数库，介于 Java 应用程序与 Database 之间。这个函数库外层是 Java 程序，负责与 Java 应用程序做沟通；而底层则是转换成另一种语言可能是 C 或 C++之类，负责与数据库做沟通。使用这个驱动程序，可以让原生代码与数据库的沟通快过 Java，但是一旦原生代码错误，则可能危害到 Server 的安全。如图 14-11 所示。

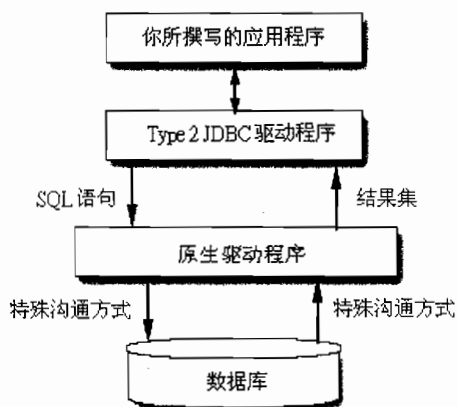


图 14-11 TYPE 2 原生 API 驱动程序概观

● TYPE 3: JDBC 通过网络的纯 Java 驱动程序

这个驱动程序是一个 3-Tier(层)的解决方案。这种驱动程序传送 JDBC 的指令到一个中介软件(Middleware)，这个中介软件再将 JDBC 的要求传送到 DBMS 当中，结果集合(ResultSet)也是通过中介软件传回到应用程序。图 14-12 是 Type 3 的驱动程序架构图：

● TYPE 4: 原生协议以及纯 Java 驱动程序

Type 4 的驱动程序是由纯 Java 所写成，并且直接与数据库做沟通。这种驱动程序不需要转换或通过其他中介软件就可以与数据库做沟通，因此性能也是最好的。图 14-13 是 Type 4 驱动程序的架构图：

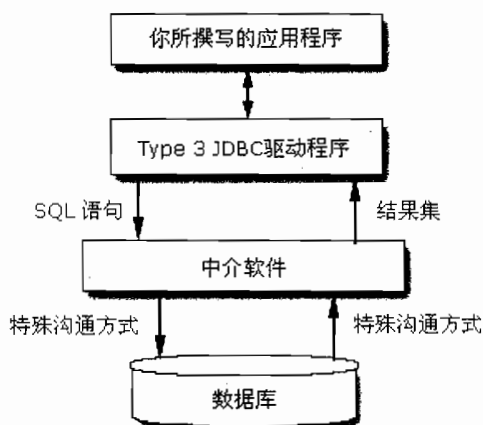


图 14-12 TYPE 3 JDBC 通过网络的纯 Java 驱动程序

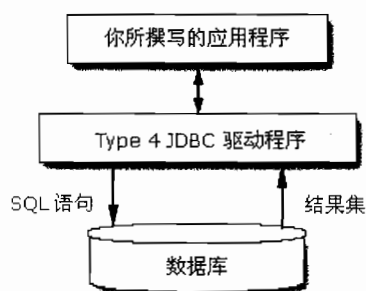


图 14-13 TYPE 4: 原生协议以及纯 Java 驱动程序

表 14-3 列出了各种不同的 JDBC 驱动程序, 以及它们支持 API 的版本和支持的数据库。笔者在撰写本章时, 大部分的驱动程序只支持 JDBC 2.0 API, 只有一些厂商推出支持 JDBC 3.0 API 的驱动程序, 其中较知名的有: DataDirect Technologies、i-net Software、MySQL 和 NetDirect。除了 MySQL 不是专门开发 JDBC 驱动程序之外, 其他三间都是专门开发 JDBC 驱动程序的公司, 不过可惜的是它们都不是免费提供的。

表 14-3

厂商名称	JDBC™ API 版本	类型	支持的数据库
Broadband Communications Solutions Corp.	3.x	<u>3</u> <u>4</u>	MS Access MS SQL Server
ATINAV, INC	3.x	<u>4</u>	MS SQL Server
Birdstep Technology	2.x	<u>4</u>	Birdstep RDM Server
Borland	3.x	<u>4</u>	JDataStore
ChipData	2.x	<u>3</u> <u>4</u>	DB2
Cipherware Ltd.	2.x	<u>4</u>	Cloudscape DB2 MySQL Oracle PostgreSQL
Cloudscape	2.x	<u>4</u>	Cloudscape
DataDirect Technologies	3.x	<u>4</u>	Oracle DB2 MS SQL Server Sybase Informix
HiT Software, Inc.	2.x	<u>4</u>	DB2
Hitachi	2.x	<u>4</u>	DABroker
HOB electronic GmbH & Co. KG	2.x	<u>4</u>	DB2
HSQL Development Group	2.x	<u>4</u>	HSQldb
i-net Software	3.x	<u>4</u>	Sybase MS SQL Server Oracle
IBM	2.x	<u>4</u>	IBM AS/400
Imaginary	2.x	<u>4</u>	mSQL
Informix Corporation	2.x	<u>4</u>	Informix

续表

厂商名称	JDBC™ API 版本	类型	支持的数据库
jTDS Project	2.x	<u>4</u>	MS SQL Server Sybase
jxDBCCon	2.x	<u>2</u> <u>4</u>	PostgreSQL
Microsoft Corporation	2.x	<u>4</u>	MS SQL Server
MySQL	3.x	<u>4</u>	MySQL
NetDirect	3.x	<u>4</u>	MS SQL Server
New Atlanta Communications	3.x	<u>4</u>	MS SQL Server
Object Industries	2.x	<u>1</u> <u>2</u> <u>3</u> <u>4</u>	DB2 JDBC MS SQL Server Oracle PostgreSQL
Open Text Corporation	2.x	<u>4</u>	BASIS
OpenBase International	2.x	<u>4</u>	OpenBase
Oracle	2.x	<u>2</u> <u>4</u>	Oracle
Pervasive Software	2.x	<u>4</u>	Pervasive.SQL
Pointbase	2.x	<u>4</u>	PointBase
PostgreSQL Development Group	2.x	<u>4</u>	PostgreSQL
Quadcap Software	2.x	<u>4</u>	Quadcap
SAP AG	2.x	<u>4</u>	SAP DB
SilverStream	2.x	<u>4</u>	Oracle
Software AG	2.x	<u>4</u>	ADABAS
Sunopsis	2.x	<u>4</u>	XML
Sybase, Inc.	2.x	<u>4</u>	Sybase
ThinkSQL Ltd	2.x	<u>4</u>	ThinkSQL
ThinWEB Technologies Inc.	2.x	<u>4</u>	MS SQL Server
Upright Database Technology	2.x	<u>4</u>	Mimer SQL
Upright Database Technology	2.x	<u>4</u>	Mimer SQL

以上的数据来自 Sun 的 Java 官方网站，如果读者有兴趣可以至：

<http://industry.java.sun.com/products/jdbc/drivers>

JSP2.0 技术手册

这里收集所有各种不同种类的 JDBC 驱动程序数据，并且还提供不错的搜索接口，方便用户来找寻合适的 JDBC 驱动程序。

14-3-2 JDBC 连接 MySQL

使用 JDBC 连接数据库存取数据时，必须要执行下列三个步骤：

- (1) 用 DriverManager 加载及注册适当的 JDBC 驱动程序；
- (2) 用 JDBC URL 定义驱动程序与数据来源之间的连接，并且建立一个连接对象；
- (3) 建立一个 SQL 陈述式对象(Statement Object)，并且利用它来执行 SQL 语句。

这三个步骤是 JDBC 连接数据库时最基本、必要的步骤。不管我们使用哪种数据库，如：Oracle、DB2、MS SQL SERVER、MySQL 等等，都一样要经过此三个步骤。

接下来笔者将依照上述三个步骤一步一步介绍，以范例的方式来让读者了解如何使用 JDBC 来存取 MySQL 的数据。

■ MySQL 驱动程序的介绍与取得

MySQL 的驱动程序称为 Connector/J，目前有两种版本，一种是只支持 JDBC 2.0 的 Connector/J 2.0，它的最新版本为 2.0.14；另一种是支持 JDBC 3.0 的 Connector/J 3.0/3.1，目前最新版本为 3.0.9 stable。它们皆可以在 MySQL 的官方网站上免费取得，网址如下：

<http://www.mysql.com/products/connector-j/index.html>

这里笔者使用 Connector/J 3.0.9 作为本书范例的 JDBC 驱动程序。接下来就正式依序介绍连接数据库的三个步骤：

■ 加载及注册 JDBC 驱动程序

JDBC 与数据库建立连接的第一个步骤为加载(load)适当的驱动程序，我们一般使用 Class.forName()来加载我们的驱动程序，以 MySQL 为例：

```
Class.forName("com.mysql.jdbc.Driver");
```

加载 MySQL 驱动程序之后，驱动程序一般都会建立一个 Driver 对象，并且经由调用 DriverManager.registerDriver()来自动注册此对象。

一般加载驱动程序时，可能会遇到的问题有下列两种：

● 加载失败，错误信息为“Class not found”

这个问题通常是因为加载驱动程序时，无法找到该 JDBC 驱动程序。解决的方法有几种：一种是将驱动程序放在 CLASSPATH 环境变量中：那假设你将 MySQL 的驱动程序 mysql-connector-java-3.0.9-stable-bin.jar 放至 C:\lib 时，你的 CLASSPATH 就应该再加上：

JSP2.0 技术手册

```
C:\lib\mysql-connector-java-3.0.9-stable-bin.jar;
```

另外一种，假若你是使用 Tomcat，你可以将 MySQL 的驱动程序放至
{Tomcat_Install}\webapps\你的站台\WEB-INF\lib

然后记得重新启动 Tomcat，这样 MySQL 的驱动程序就能顺利加载。

● 加载失败，错误信息为“Driver not found”

这个错误信息表示你没有用 DriverManager 类来注册 JDBC 驱动程序。有时候当我们用 Class.forName()来加载驱动程序时，会碰到 JDBC 规范与某些 JVM 产生问题。当遇到这种情况时，我们可以把加载程序改为：

```
Class.forName("com.mysql.jdbc.Driver").newInstance( );
```

■ JDBC URL 定义驱动程序与数据来源之间的连接

与数据库连接的第二步骤为：将驱动程序连接到数据来源的 JDBC URL。

JDBC URL 的标准语法如下：

```
<protocol>:<subprotocol>:<data source identifier>
```

由上述可知，JDBC URL 分为三个部分：

<protocol>：主要通讯协议

<subprotocol>：次要通讯协议，即驱动程序的名称

<data source identifier>：数据来源

假若我们使用 JDBC-ODBC bridge driver 来连接一个名为 mydb 的本地端数据库时，我们可以利用下面 JDBC URL 格式来连接数据库：

```
jdbc:odbc:mydb?user=user_name&password=user_password
```

而 MySQL 的 JDBC URL 如下列的格式：

```
jdbc:mysql://[hostname][:port]/[dbname][?param1=value1][&param2=value2].....
```

hostname 表示联机主机名称，也可以为一个有效 IP 地址；port 为连接数据库所使用的端口；dbname 就是数据库的名称；最后就是一些参数的设定(param=value)，不过必须注意到，别忘记在 dbname 之后加上问号“？”，再接你所要设定之参数值。现在笔者连接之前章节所建立的 sample_db 数据库，因此我们 JDBC URL 可以表示为如下所示：

```
jdbc:mysql://localhost:3306/sample_db?user=root&password=your_password
```

上述 JDBC URL 表示：我们将连接一个本地端且连接端口为 3306 的 MySQL 服务器，并且指定连接 sample_db 数据库，问号后面所接的参数代表登录此数据库的名称和密码，除了 user 和 password 两个参数之外，MySQL 驱动程序还提供其他常见可以设定的参数，都列在表 14-4 中：

JSP2.0 技术手册

表 14-4

参 数	说 明	默认值
user	联机时, 用户名称	None
password	联机时, 用户密码	None
autoReconnect	如果第一次联机失败时, 驱动程序是否会重新联机(true/false)	False
maxReconnects	如果 autoReconnect 功能启动时, 会尝试重新联机的次数	3
initialTimeout	如果 autoReconnect 功能启动时, 会尝试重新联机的间隔时间(以秒计算)	2
maxRows	传回之行的最大数(0 表示传回所有的行)	0
useUnicode	处理 string 时, 驱动程序是否应该使用 Unicode 字体编码(true/false)	False
characterEncoding	如果 useUnicode 为 true, 在处理 string 时, 驱动程序应该使用何种字体编码(GB2312/UTF-8/ ...)	None
relaxAutocommit	设定是否采用自动提交(Auto-commit)	False
capitalizeTypeNames	数据定义(Meta Data)的名称以大写表示	False

上述介绍的参数是比较常用到的, 在 Connector/J 2.0.14 和 Connector/J 3.0.9 又新增一些参数, 如 connectTimeout、socketTimeout、useTimezone、continueBatchOnError、strictUpdates 和 useSSL 等等。如果读者有兴趣, 可以自行查看 Connector/J 中的 README 文件。

■ 建立一个连接对象(Connection Object)

之前完成加载驱动程序并且建立一个 JDBC URL 将驱动程序连接到数据来源, 现在可以利用 DriverManager.getConnection() 来建立一个连接对象, 语法如下所示:

```
String url =
    "jdbc:mysql://localhost:3306/sample_db?user=root&password=your_password"
Connection con = DriverManager.getConnection( url );
```

上述是笔者利用 JDBC 来连接之前建立的 sample_db 数据库, 并且利用 DriverManager.getConnection() 取得连接对象。getConnection() 还有下列的用法:

```
String url = "jdbc:mysql://localhost:3306/sample_db" ;
String user = "root" ;
String password = "your_password" ;
Connection con = DriverManager.getConnection(url, user, password) ;
```

这两种方法的执行结果都是一样的, 因此读者可以自由选择习惯的用法。

■ 建立一个 SQL 陈述式对象(Statement Object)

SQL 陈述式对象主要用来执行 SQL 语句。当我们顺利取得连接对象时, 可以利用它来传送

SQL 语句至数据库服务器，执行 SQL 语句的工作。

我们可以使用 `Connection.createStatement()` 来建立一个新的陈述式对象。下面产生一个名为 `stmt` 的陈述式对象，其做法如下：

```
Statement stmt = con.createStatement( );
```

■ 使用 SQL 陈述式对象来执行 SQL 语句

一般而言，我们只会用到 `Statement` 类中三种语句执行方法，以下介绍这三种常见的方法：

● `executeQuery()`

`executeQuery()` 方法是以一个 SQL 语句作为自变量，回传一个 `ResultSet` 对象格式的查询结果。`executeQuery()` 方法主要用于查询数据库，底下为一个简单的范例：

```
Statement stmt = con.createStatement( );
String query = "SELECT * FROM employee";
ResultSet rs = stmt.executeQuery(query);
```

`executeQuery()` 方法会回传一个 `ResultSet` 对象，这是我们之前所没有提到的新对象，接下来简单介绍 `ResultSet` 对象的功用。

结果集合对象(`ResultSet`)一般来说，当你向数据库查询一笔或多笔数据的时候，数据库会回传一个结果集合到 JDBC 中，这个结果集合也就是我们要探讨的 `ResultSet` 对象。之前的例子当中，我们声明一个 `ResultSet` 类型的变量 `rs` 来取得查询后的数据，读者可以将它想成是数据表(table)中一笔一笔的记录(record)，分别记录在 `ResultSet` 对象之内。

若要取得这一笔一笔的纪录，就要使用 `rs.next()` 方法。`ResultSet` 对象内有一个隐藏的光标，在一开始的时候是指向**第一笔数据之前**。若你要将光标向下移动，就必须利用 `rs.next()` 将指向记录的光标向下移动，每调用一次 `rs.next()` 光标便会向下移动一笔。在我们的例子中，我们是查询以取得单笔数据，因此只调用一次 `rs.next()` 就能取得我们想要的数据集，若是你想要取得查询后的多笔记录，则必须利用 `while` 循环一笔一笔地取得，就像这样：

```
while(rs.next()) {
    //在这里取得这笔数据的结果 ...
    rs.getString(1);
    rs.getInt(2);
}
```

当我们把光标指到想要的记录后，再来就是取得每个字段的数据。`ResultSet` 对象提供一系列的 `getXXX()` 方法，分别来取得不同类型的数据，而 XXX 即为类型名称，如：`String`、`int`、`Date` 等等。

● `executeUpdate()`

`executeUpdate()` 主要用于两方面：一为执行如：`CREATE TABLE`、`DROP TABLE` 和

ALTER TABLE 等 SQL DDL 语句; 二为执行如: INSERT、DELETE 和 UPDATE 等 SQL DML 语句。executeUpdate() 方法所传回的数值都为整数, 其代表的意思是更新的数据笔数。但执行 CREATE TABLE 和 DROP TABLE 时, executeUpdate() 方法所传回的数值都为零。接下来也举个简单的范例:

```
Statement stmt = con.createStatement();
String upd = "INSERT INTO employee (employee_id, last_name, first_name,
                                     birth, sex, email) VALUES ( null, '
                                     林', '上杰', '1978/12/27', 'M',
                                     'abc@abc.com' )";
stmt.executeUpdate(upd);
```

● execute()

execute() 方法主要用在如果你不知道正要执行的 SQL 是查询还是更新时, 就可以利用这个方法。这通常会发生在应用程序执行动态建立 SQL 语句的场合, 如果执行查询语句时, 那么 execute() 会传回 true; 反之, 则传回 false。

■ JDBC 连接 MySQL 范例

上述已经介绍完 JDBC 连接数据库的三步骤, 现在笔者就做一个总结, 用 JDBC 连接上一章所建立的 sample_db 数据库, 并且新增一笔数据至 employee 数据表中, 完整的程序如下:

```
try
{
    Class.forName("com.mysql.jdbc.Driver").newInstance();
    con = DriverManager.getConnection("jdbc:mysql://localhost:3306/
                                     sample_db?user=root&password
                                     =browser");

    stmt = con.createStatement();
}
catch(SQLException sqle)
{
    out.println("SQL Exception :"+sqle);
}

String upd = "INSERT INTO employee ( employee_id, last_name, first_name,
                                     birth, sex, email) VALUES ( null,
                                     'lin', 'mike', '1978/12/27', 'M',
                                     'abc@abc.com' )";

stmt.executeUpdate(upd);
```

上述惟一要说明的是, 如果在加载驱动程序或取得连接对象有错误时, 此时会产生 SQLException 对象。通常我们取得 SQLException 对象, 可以利用它的 getMessage() 来取得 SQL 的错误信息, 知道错误发生的原由, 这样子有助于程序代码的调试与侦测。

SQLException 类有一特性, 那就是当有一连串的异常事件产生时, 都会归结到 SQLException。例如: 如果一个交易在执行过程中有好几处错误发生, 那么这些错误都会被抛

入到这个类内。下面有一个好用的范例程序，能够清楚完整显示出错误信息。

```
try {
    .....
}
catch(SQLException sqle)
{
    while( sqle != null )
    {
        System.err.println("SQLState: " + sqle.getSQLState( ) ) ;
        System.err.println("Error Code: " + sqle.getErrorCode( ) ) ;
        System.err.println("Message: " + sqle.getMessage( ) ) ;
        sqle = sqle.getNextException( ) ;
    }
}
```

■ Java 数据类型和 SQL 数据类型

JDBC 定义了 Java 数据类型对应到 SQL 的数据类型的标准。如表 14-5 所示：

表 14-5

Java 类型	SQL 类型	JDBC 存取方法
boolean	BIT	getBoolean()
byte	TINYINT	getByte()
short	SMALLINT	getShort()
int	INTEGER	getInt()
long	BIGINT	getLong()
float	REAL	getFloat()
double	FLOAT	getDouble()
double	DOUBLE	getDouble()
java.math.BigDecimal	DECIMAL	getBigDecimal()
java.math.BigDecimal	NUMERIC	getBigDecimal()
java.lang.String	CHAR	getString()
java.lang.String	VARCHAR	getString()
java.lang.String	LONGVARCHAR	getString()
java.util.Date	DATE	getDate()
java.sql.Time	TIME	getTime()
java.sql.Timestamp	TIMESTAMP	getTimestamp()
byte[]	BINARY	getBytes()
byte[]	VARBINARY	getBytes()

续表

Java 类型	SQL 类型	JDBC 存取方法
byte[]	LONG VARBINARY	getBytes()
java.sql.Blob	BLOB	getBlob()
java.sql.Clob	CLOB	getClob()
java.sql.Array	ARRAY	getArray()
java.sql.Ref	REF	getRef()
java.sql.Struct	STRUCT	getTimestamp()

从表格可以知道二者之间的对应必非 1 对 1 的映像(mapping), 例如: SQL 类型的 CHAR、VARCHAR 和 LONGVARCHAR 都是对应到 Java 类型的 String, 因此我们可以使用 getString() 来取得此三种 SQL 类型的数据。

14-4 JDBC 连接 MySQL 的中文问题

本章将举几个范例, 让读者了解如何从网页窗体取得信息, 并且将信息存入 MySQL 中。这个过程其实不算困难, 但是初学者常遇到中文问题: 有某些中文字存入 MySQL 时, 会发生不寻常的错误或者中文字会变成乱码, 无法正确储存。因此本章另一个重点是: 讨论如何解决上述的中文问题, 让读者能够顺利使用 JDBC 与 MySQL。

接下来的所有范例, 笔者将会使用之前所建立的 employee 数据表来作为范例。首先第一个范例包含两个部分, 第一个部分是网页窗体, 它主要是让用户输入想要新增的姓和名, 文件名为 *Mysql.html*; 第二个部分是 JSP 网页, 它主要是将由 *Mysql.html* 所接收的数据存入至 employee 数据表中。

■ Mysql.html

```
<html>
<head>
<title>mysql</title>
<meta http-equiv="Content-Type" content="text/html; charset=GB2312">
</head>

<body>
<form name="form" method="post" action="mysql.jsp">
  <p>姓: <input name="last_name" type="text" id="last_name"></p>
  <p>名: <input name="first_name" type="text" id="first_name"></p>
  <p>
    <input type="submit" value="传送">
    <input type="reset" value="取消">
  </p>
</form>
</body>
</html>
```

JSP2.0 技术手册

Mysql.html 网页主要有两个本文字段：姓和名，当按下【传送】时，会将信息传送到 *Mysql.jsp* 做处理。*Mysql.html* 的执行结果如图 14-14 所示：



图 14-14 Mysql.html 的执行结果

接下来主要处理数据的 JSP 网页，文件名为 *Mysql.jsp*：

■ *Mysql.jsp*

```
<%@ page import="java.sql.*" %>
<%@ page contentType="text/html; charset=GB2312" %>

<html>
<head>
  <title>CH14 - Mysql.jsp</title>
</head>
<body>

<h2>将信息存入 Mysql 中</h2>
<%
  Connection con = null;
  Statement stmt = null;
  Statement stmt1 = null;
  ResultSet rs = null;

  request.setCharacterEncoding("GB2312");

  String employee_id = null;
  String last_name = request.getParameter("last_name");
  String first_name = request.getParameter("first_name");
  String birth = "1978/12/11";
  String sex = "F";
  String email = "aaa@asdf.com";

  String new_last_name = "";
```

JSP2.0 技术手册

```
String new_first_name = "";
%>
从 mysql.html 接收到的信息如下: <br>
姓: <%= last_name %>
名: <%= first_name %><br><br>
<%
try
{
    Class.forName("com.mysql.jdbc.Driver").newInstance();
    con = DriverManager.getConnection("jdbc:mysql://localhost:3306/
        sample_db?
        user= root&password=browser&useUnicode
        =true&characterEncoding=GB2312");
    stmt = con.createStatement();

    String upd = "INSERT INTO employee(employee_id, last_name,
        first_name, birth, sex, email) VALUES
        ('+employee_id+', '"+last_name+',
        '"+first_name+', '"+birth+', '"+sex+',
        '"+email+"'");

    stmt.executeUpdate(upd);
    stmt1 = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
        ResultSet.CONCUR_UPDATABLE);
    String query = "SELECT * FROM employee";
    rs = stmt1.executeQuery(query);

    rs.last( );
    new_last_name = rs.getString("last_name");
    new_first_name = rs.getString("first_name");

    stmt.close();
    stmt1.close();
    con.close();
}
catch(SQLException sqle)
{
    out.println("sqle="+sqle);
}
finally
{
    try {
        if(con != null)
        {
            con.close( );
        }
    }
    catch(SQLException sqle)
    {
        out.println("sqle="+sqle);
    }
}
```

```
%>
    从 employee 取出最新新增的姓名: <br>
    新增姓名: <%= new_last_name+new_first_name %><br>
</body>
</html>
```

Mysql.jsp 程序主要包含几个部分, 第一个部分是变量的声明, 如下所示:

```
Connection con = null;
Statement stmt = null;
Statement stmt1 = null;
ResultSet rs = null;
```

笔者声明一个 Connection 对象 con、两个 Statement 对象 stmt 和 stmt1 和一个 ResultSet 对象 rs。

第二部分是接收由 *Mysql.html* 所传来的信息, 并且确认能够接收到正确的中文信息。其主要程序代码如下:

```
<%
    request.setCharacterEncoding("GB2312");

    String employee_id = null;
    String last_name = request.getParameter("last_name");
    String first_name = request.getParameter("first_name");
    String birth = "1978/12/11";
    String sex = "F";
    String email = "aaa@asdf.com";

    String new_last_name = "";
    String new_first_name = "";
%>
    从 mysql.html 接收到的信息如下: <br>
    姓: <%= last_name %>
    名: <%= first_name %><br><br>
```

从窗体接收中文字时, 必须要设定传送时所要用的编码, 例如: 本范例是使用 GB2312 来做编码的, 因此我们必须加上 request.setCharacterEncoding("GB2312");, 否则无法正确接收到中文。

注意

setCharacterEncoding(String encode)方法是 Servlet 2.3、JSP 1.2 新增的方法。

第三部分是 JDBC 连接 MySQL, 程序代码如下:

```
try
{
    Class.forName("com.mysql.jdbc.Driver").newInstance();
    con = DriverManager.getConnection("jdbc:mysql://localhost:3306/
        sample_db?
```

JSP2.0 技术手册

```

        user=root&password=browser&useUnicode=true&characterEncoding
        =GB2312");
        stmt = con.createStatement();
        .....
    }
    catch(SQLException sqle)
    {
        out.println("sqle="+sqle);
    }

```

JDBC 连接 MySQL 的使用方法在上一节已经详细说明过，这里必须要注意的就是在连接 MySQL 的参数中，还要加上 useUnicode=true 和 characterEncoding=GB2312，假若没有加上这两个参数，则无法正确储存中文。

最后一个部分就是执行新增数据的 SQL 语句，并且再从数据库中取得最近一笔新增的信息显示至网页上，用来确认新增中文无误，程序代码如下：

```

<%
    try {
        .....
        String upd = "INSERT INTO employee(employee_id, last_name, first_name,
            birth, sex, email) VALUES ('+employee_id+',
            '+last_name+', '+first_name+', '+birth+',
            '+sex+', '+email+')";
        stmt.executeUpdate(upd);
        stmt1 = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
            ResultSet.CONCUR_UPDATABLE);
        String query = "SELECT * FROM employee";
        rs = stmt1.executeQuery(query);
        rs.last();

        new_last_name = rs.getString("last_name");
        new_first_name = rs.getString("first_name");
        ..... 略
    }
    catch (Exception e) {
        ..... 略
    }
%>
    从 employee 取出最新新增的姓名: <br>
    新增姓名: <%= new_last_name+new_first_name %><br>

```

这里我利用 Statement 对象的 executeUpdate() 方法来执行新增信息的 SQL 语句；用 executeQuery() 方法来执行查询数据的 SQL 语句，它会回传一个 ResultSet 对象。这里笔者使用到 JDBC 2.0 API，这部分留到后面的章节再来详细说明它们。

最后使用完 Statement 和 Connection 对象之后，别忘了要释放它们的资源。

```

<%
    try {
        ..... 略
        stmt.close();
    }
    catch (Exception e) {
        ..... 略
    }
%>

```

JSP2.0 技术手册


```
stmt1.close();
con.close();
}
catch(SQLException sqle)
{
    out.println("sqle="+sqle);
}
finally
{
    try {
        if(con != null)
        {
            con.close();
        }
    }
    catch(SQLException sqle)
    {
        out.println("sqle="+sqle);
    }
}
%>
```

Mysql.jsp 执行结果如图 14-15 所示:

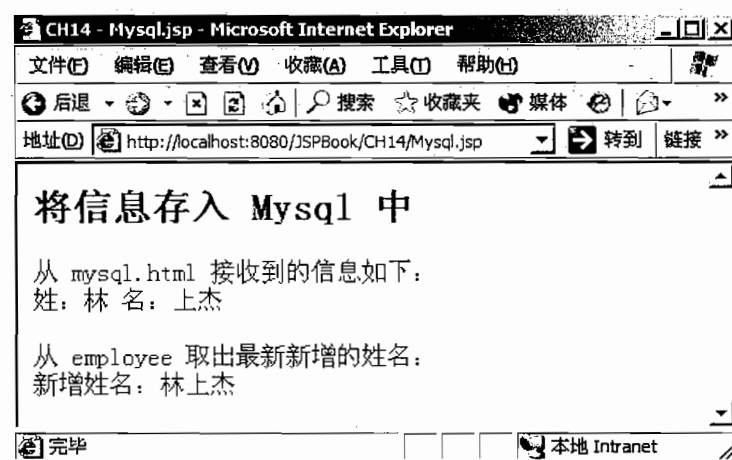


图 14-15 Mysql.jsp 的执行结果

目前 MySQL 的 employee 数据表内容应该如图 14-16 所示。

到目前为止, 一般的中文字应该都能够正确储存起来, 接下来看几个特殊的中文字, 当遇到这些字时, 会发生什么问题, 又应当如何解决它们?

```

mysql> USE sample_db
Database changed
mysql> SELECT * FROM employee;
+-----+-----+-----+-----+-----+-----+
| employee_id | last_name | first_name | birth      | sex | email      |
+-----+-----+-----+-----+-----+-----+
| 1 | 林 | 上杰 | 1978-12-11 |     | aaa@asdf.com |
+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql>

```

图 14-16

14-5 PreparedStatement

预编语句(PreparedStatement)接口是继承 Statement 接口的, 它和 Statement 接口有两个不同之处:

- 第一: 预编语句是包含已编译好的 SQL 语句; 而 Statement 的 SQL 语句是当程序要执行时, 才会去编译它们。
- 第二: 预编语句中的 SQL 语句具有一个或多个参数, 每个参数用问号“?”替代。每个问号的值必须在执行之前利用 setXXX 方法来设定它所代表的值。如下面这段程序代码:

```

PreparedStatement stmt = conn.prepareStatement(" INSERT INTO myTable(
    name , "+"number , sex , phone ) VALUES
    ( ?,?,?,? )");

stmt.setString(1, "MIKE");
stmt.setInt(2,1234);
stmt.setString(3, "Male");
stmt.setString(4, "2314-4567");

```

我们先声明一 PreparedStatement 对象 stmt, 它包含一个 SQL 语句:

```
INSERT INTO myTable(name , number , sex , phone ) VALUES ( ?,?,?,? )
```

这是一个 INSERT 数据的 SQL 语句, 它和一般 Statement 中的 SQL 语句不一样, 它的 VALUES 后面接的参数皆用问号“?”替代, 表示它们都是变量, 第一个问号对应到 name 字段; 第二个问号就对应到 number 字段, 依此类推。

当真正要执行时, 才利用 setXXX 方法来指定问号的值。这里附带一点: setXXX 方法中的 XXX 是指数据类型, 假设你要设定 name 字段的值, 而 name 字段是 VARCHAR 或 CHAR 数据类型时, 我们就必须使用 setString()方法; 若要设定 INTEGER 数据类型的 number 字段, 就要用 setInt()方法来设定其值。

```
stmt.setString(1, "MIKE");  
stmt.setInt(2, 1234);  
stmt.setString(3, "Male");  
stmt.setString(4, "2314-4567");
```

这四行程序分别设定这四个问号所代表的值。setXXX 方法中第一个参数：设定第几个问号；第二个参数：要设定的值为何。例如：setString(1, "MIKE")，就表示设 name = MIKE。

由于 PreparedStatement 对象中的 SQL 语句已预先编译过，所以其执行速度要快于 Statement 对象。因此经常执行的 SQL 语句都是用 PreparedStatement 对象，以提高效率。

■ 建立 PreparedStatement 对象

以下面的程序代码（其中 con 是 Connection 对象）建立一个 PreparedStatement 对象，而 PreparedStatement 中的 SQL 语句有两个参数：

```
PreparedStatement pstmt = con.prepareStatement(  
    "UPDATE table SET m = ? WHERE x = ?" );
```

pstmt 对象包含 SQL 语句 "UPDATE table SET m = ? WHERE x = ?", 它已经传送给 DBMS，并且做好准备等待执行。

■ 设置参数

在执行 PreparedStatement 中的 SQL 语句之前，必须先设定每个 ? 参数的值，这可以利用 setXXX 方法来完成，其中 XXX 是指该参数的数据类型。例如：如果参数具有 Java 类型 long，则使用的方法就是 setLong。setXXX 方法的第一个参数是选择表格中的字段，第二个参数是设置给该字段的值。例如将第一个字段设为 123456789，第二个字段设为 100000000：

```
pstmt.setLong(1, 123456789);  
pstmt.setLong(2, 100000000);
```

一旦设定语句的参数值后，就可以多次执行该语句，直到调用 clearParameters() 方法将它清除为止。

利用 pstmt（前面建立的 PreparedStatement 对象），以下程序示范如何设置两个参数的值，并执行同一 SQL 语句 10 次。如上所述，为做到这一点，数据库不能关闭 pstmt。在该范例中，第一个字段被设置为 "Hi" 并保持为常数，在 for 循环中，每次都第二个字段设置为不同的值：从 0 开始到 9 结束。

```
pstmt.setString(1, "Hi");  
  
for (int i = 0; i < 10; i++)  
{  
    pstmt.setInt(2, i);  
    int rowCount = pstmt.executeUpdate();  
}
```

14-6 CallableStatement

所谓的预储程序(Stored Procedure)是指:在你执行应用程序之前,预储程序就已经内建在数据库中;在执行应用程序期间,直接以名称存取预储程序。预储程序具有以下优点:

- (1) 由于大多数的数据库都会事先将预储程序编译好暂存在数据库中,因此,预储程序的执行速度会比在程序中调用 SQL 语句要快。
- (2) 预储程序的语法有错误时,可以在编译期间解决,而非在执行期间。
- (3) Java 开发人员只须要知道预储程序的名称、输入和输出,就能够顺利执行工作。至于预储程序如何存取数据表格、表格数据类型等等,皆与 Java 无关。

注意

Mysql 4.0x 尚未支持预储程序,根据 Mysql 的官方数据: Mysql 5.0 才会支持预储程序。目前大部分商业型的数据库都支持预储程序。

当预储程序被调用时,传递的自变量就是预储程序所需的变量,笔者以 Oracle 举例说明:

```
CREATE OR REPLACE PROCEDURE count_interest
( id IN INTEGER, bal OUT FLOAT ) IS
BEGIN
    SELECT balance INTO bal FROM account WHERE account_id = id;
    bal := bal + bal * 0.02;
    UPDATE account SET balance = bal WHERE account_id = id;
END;
```

此预储程序有两个自变量值: INTEGER 类型的 id 和 FLOAT 类型的 bal。其中 id 的参数有一个 IN 的关键字,表示 id 参数能用 setId()方法设定其值,但是不能用 getId()方法取得此值; bal 参数为 OUT 的关键字,表示 bal 参数能用 getBal()方法取得此值,但是不能用 setBal()方法设定其值。除此之外,还有另一种关键字 IN OUT,表示参数能使用 getXXX、setXXX 方法来取得或设定此值。

这个预储程序执行两个 SQL 语句,并且做一次计算:第一步先取得目前的余额;第二步将余额增加 2%;第三步则更新余额。接下来就是使用 JDBC 的 CallableStatement 来调用此预储程序:

```
try {
    Connection conn;
    CallableStatement callstmt;
    int i;

    conn.setAutoCommit(false);

    callstmt = conn.prepareCall("{call count_interest [ (? , ? ) ] }");
    callstmt.registerOutParameter(2, java.sql.Types.FLOAT);
```

JSP2.0 技术手册

```
for ( i = 1 ; i < accounts.length ; i++)
{
    callstmt.setInt(1, accounts[i].getId( ));
    callstmt.execute( );
    System.out.println("New balance: " + callstmt.getFloat(2) )
}
conn.commit( );
callstmt.close( );
conn.close( );
}
```

一般来说, 使用 CallableStatement 来调用预储程序, 有五个步骤:

第一步, 建立预储程序;

笔者已经先建立一个名为 count_interest 的预储程序。

第二步, 建立 CallableStatement 对象;

```
CallableStatement callstmt;
callstmt = conn.prepareCall("{call count_interest [ ( ? , ? ) ] }");
```

使用 Connection 对象的 prepareCall() 来调用预储程序。JDBC 提供独立于数据库与预储程序之外的调用语法: 转义语法(escape syntax), 其形式为:

```
{call [schema.][package.]procedure_name [ ( ? , ? , ..... ) ] }
```

其中问号 “?” 代表预储程序的自变量。

第三步, 注册所有 OUT 的参数;

根据 count_interest 预储程序, 它第二个自变量 bal 有 OUT 关键字, 因此必须使用 registerOutParameter() 来注册输出的 bal 参数, 如下:

```
callstmt.registerOutParameter(2, java.sql.Types.FLOAT);
```

上述表示 count_interest 预储程序的第二个自变量为 OUT, 类型为 FLOAT。

第四步, 设定所有 IN 的参数;

根据 count_interest 预储程序, 第一个自变量 id 有 IN 关键字, 而且 id 的类型为 INTEGER, 所以设定的方法如下:

```
callstmt.setInt(1, accounts[i].getId( ));
```

第五步, 执行 CallableStatement。

最后执行 CallableStatement 的方法如下:

```
callstmt.execute( );
```

执行之后, 可以使用 getXXX() 的方法来取得数据, 例如: 取得最后计算后的余额总数。如下:


```
callstmt.getFloat(2);
```

14-7 JDBC 2.0 介绍与使用

若想从查询的结果中只显示第三十笔数据时，依照 JDBC 1.0 的做法只能执行 `ResultSet.next()` 30 次，才能得到所要的数据。假若我想显示第 100000 笔到第 100010 笔的数据时，那不就要先执行十万次无意义的动作，才能够顺利取得我想要的数据库。

不过 JDBC 2.0 已经解决上述的问题，并且还提供一些方便好用的功能，在接下来的部分会有详尽的说明。

14-7-1 JDBC 2.0 介绍

现在开始进入主题，首先当然要先介绍 JDBC 2.0 和之前 1.0 有哪些不同之处。JDBC 2.0 API 和 1.0 有四项差异较大的部分：

- (1) `ResultSet` 对象中的光标(Cursor)能够上下自由移动；
- (2) 能直接使用 Java 程序语言来更新数据库表格的内容，而不需要写 SQL 语法；
- (3) 可以一次传送许多 SQL 语句到数据库执行——批处理(Batch)；
- (4) 进阶数据类型——BLOB 和 CLOB。

接下来的小节将为读者介绍上述四项差异较大的部分。

14-7-2 `ResultSet` 对象中的光标能够上下自由移动

首先我们看下面这个范例：

```
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,  
                                       ResultSet.CONCUR_READ_ONLY);  
ResultSet srs = stmt.executeQuery("SELECT COF_NAME, PRICE FROM COFFEES");
```

JDBC 1.0 时，只要 `connection.createStatement()` 就能够顺利取得 `Statement` 对象。不过 JDBC 2.0 的 `createStatement()` 方法多了两个参数值，需要用户自行定义它。

```
public Statement createStatement(int resultSetType, int  
                                resultSetConcurrency) throws SQLException
```

第一个参数 `resultSetType` 用来设定 `ResultSet` 对象中的光标是否能够上下自由移动，它的值只能有三种，`TYPE_FORWARD_ONLY`、`TYPE_SCROLL_SENSITIVE` 或是 `TYPE_SCROLL_INSENSITIVE`。若设为第一种，那就表示 `ResultSet` 对象和 JDBC 1.0 的没差别，都只能使用 `next()` 方法；而后两者都可以让 `ResultSet` 对象中的光标能够随心所欲地上下移动，不过它们两者最大的差别在于：当 `ResultSet` 对象中的值有改变时，`TYPE_SCROLL_SENSITIVE` 能够取得改变后的值，而 `TYPE_SCROLL_INSENSITIVE` 不能。

第二个参数 `resultSetConcurrency`，主要设定 `ResultSet` 对象是只读(read-only)还是可改

JSP2.0 技术手册

变的(updatable), 它可能的值只有两种, CONCUR_READ_ONLY 或是 CONCUR_UPDATABLE。若设为 CONCUR_READ_ONLY, ResultSet 对象和 JDBC 1.0 的功能一样; 若为 CONCUR_UPDATABLE, 那表示 ResultSet 对象可以直接执行数据库的新增、修改和移除的功能。接下来看一个范例:

```
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                     ResultSet.CONCUR_READ_ONLY);
ResultSet srs = stmt.executeQuery("SELECT COF_NAME, PRICE FROM COFFEES");
while (srs.next())
{
    String name = srs.getString("COF_NAME");
    float price = srs.getFloat("PRICE");
    System.out.println(name + "      " + price);
}
```

其结果可能为:

```
Colombian      7.99
French_Roast   8.99
Espresso       9.99
Colombian_Decaf 8.99
French_Roast_Decaf 9.99
```

这与 JDBC 1.0 没有什么差别。接着再看下一个范例:

```
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                                     ResultSet.CONCUR_READ_ONLY);
ResultSet srs = stmt.executeQuery("SELECT COF_NAME, PRICE FROM COFFEES");
srs.afterLast();
while(srs.previous())
{
    String name = srs.getString("COF_NAME");
    float price = srs.getFloat("PRICE");
    System.out.println(name + "      " + price);
}
```

其结果可能为:

```
French_Roast_Decaf 9.99
Colombian_Decaf   8.99
Espresso          9.99
French_Roast      8.99
Colombian         7.99
```

这和之前的结果不一样的地方在于顺序都颠倒过来, 这是因为使用 srs.afterLast() 这个方法主要是将光标移到最后一笔数据之后, 然后再调用 srs.previous() 判断前面是否还有数据, 若有, 就把结果显示出来; 若没有, 程序就停止。

ResultSet 对象 srs 除了新增 afterLast() 和 previous() 方法之外, 还新增了许多方法, 我们就直接用范例来说明, 读者很快就能了解到每个方法的功能和使用方式。

```
srs.absolute(1) // 表示光标在第一笔数据, 即 srs.first()
srs.absolute(-1) // 表示光标在最后一笔数据, 即 srs.last()
srs.absolute(4) // 表示光标在第四笔数据
```

假若目前 ResultSet 有 1000 笔数据, 若要取得第 997 笔数据时, 如下:

```
srs.absolute(-4) // 表示光标在第 997 笔数据
```

还有一个方法与 absolute() 很像, 那就是 relative(), 我们直接来看范例:

```
srs.absolute(5) // 表示光标在第五笔数据
.....
srs.relative(-3) // 表示光标目前在第二笔数据
.....
srs.relative(1) // 表示光标目前在第三笔数据
```

看完上述范例是不是觉得使用起来非常的简单、好用, 因此可以使用 absolute() 和 relative() 方法来任意取得我们所想要的记录。

14-7-3 直接使用 ResultSet 对象执行更新数据的动作:

假若我们要更新数据库中的最后一笔数据: French_Roast_Decaf, 则 JDBC 1.0 的做法可能为:

```
Statement stmt = con.createStatement();
stmt.executeUpdate("UPDATE COFFEES SET PRICE = 10.99" +
    "WHERE COF_NAME = French_Roast_Decaf ");
```

JDBC 2.0 的做法可以为:

```
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
    ResultSet.CONCUR_UPDATABLE);
ResultSet uprs = stmt.executeQuery("SELECT COF_NAME, PRICE FROM COFFEES");
uprs.last();
uprs.updateFloat("PRICE", 10.99);
uprs.updateRow();
```

利用 ResultSet 对象的 updateXXX() 方法直接更改 ResultSet 对象中的数据, 不过这里读者须要注意的地方就是一开始在声明 Statement 对象时, 这两个参数必须分别设为: TYPE_SCROLL_SENSITIVE 和 CONCUR_UPDATABLE。当执行到 uprs.updateFloat("PRICE", 10.99) 时, 数据库的数据还未做完更新动作, 直到调用 uprs.updateRow() 方法, 才真正执行 uprs.updateFloat("PRICE", 10.99)。接下来再看一个范例:

```
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
    ResultSet.CONCUR_UPDATABLE);
ResultSet uprs = stmt.executeQuery("SELECT COF_NAME, PRICE FROM COFFEES");
uprs.last();
uprs.updateFloat("PRICE", 10.99);
uprs.cancelRowUpdates();
uprs.updateFloat("PRICE", 10.79);
uprs.updateRow();
```

JSP2.0 技术手册

这个范例主要说明 `cancelRowUpdates()` 方法取消之前更新的动作, 最后 `PRICE` 被改为 10.79, 而不是 10.99。之前都使用 `ResultSet` 对象做更新的动作, 接下来就介绍新增的动作。

■ 直接使用 `ResultSet` 对象执行新增数据的动作

假若我们要新增一笔数据到数据库中, JDBC 1.0 的做法可能为:

```
stmt.executeUpdate("INSERT INTO COFFEES " + "VALUES ('Kona', 150, 10.99, 0, 0)");
```

JDBC 2.0 的做法可以为:

```
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                     ResultSet.CONCUR_UPDATABLE);
ResultSet uprs = stmt.executeQuery ("SELECT * FROM COFFEES");

uprs.moveToInsertRow();
uprs.updateString(1, "Kona");
uprs.updateInt(2, 150);
uprs.updateFloat(3, 10.99);
uprs.updateInt(4, 0);
uprs.updateInt(5, 0);
uprs.insertRow();
```

调用 `uprs.moveToInsertRow()` 将光标移到要新增的那一笔数据, 然后利用 `uprs.updateXXX()` 将数据根据 `XXX` 类型新增到正确的字段, 最后再调用 `uprs.insertRow()` 方法执行新增数据的动作。

■ 直接使用 `ResultSet` 对象执行删除数据的动作

JDBC 2.0 提供 `ResultSet` 对象一个方便删除数据的方法:

```
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                     ResultSet.CONCUR_UPDATABLE);
ResultSet uprs = stmt.executeQuery ("SELECT * FROM COFFEES");

uprs.absolute(4);
uprs.deleteRow();
```

这样就直接把 `ResultSet` 对象中第四笔数据删除了。

14-7-4 批处理(Batch)

JDBC 2.0 API 提供批处理(batch)的功能, 它可以让一个 `Statement` 同时执行多个 SQL 语句, 以提高执行性能。下面来看一个批处理的范例程序:

```
Statement stmt = con.createStatement();
int[] l rows;
stmt.addBatch("INSERT INTO employee VALUES ('Mike', 123456)");
stmt.addBatch("INSERT INTO office VALUES ('employee', 'Taipei')");
```

JSP2.0 技术手册

```
rows = stmt.executeBatch( );
```

上述范例说明了 batch 的用法,当执行 executeBatch()时,它会同时新增 employee 和 office 数据表的数据,最后回一个 int 类型的数组,表示修改过数据行的数目。

不过这里有一个地方须要注意的:即在上述的范例中,我们默认自动认可(auto-commit),因此每一次的更新都会自动认可。假若批处理中有错误发生时,则在错误发生前所有更新的动作将被执行;而错误发生后的更新动作将无法执行。因此,可以使用 executeBatch()抛出的 BatchUpdateException 事件中的 getUpdateCounts()方法,来取得有多少 SQL 语句执行成功。如下所示范例:

```
Statement stmt = con.createStatement( );
int[ ] rows;
stmt.addBatch(" 1 .....");
stmt.addBatch(" 2 .....");
stmt.addBatch(" 3 .....");
stmt.addBatch(" 4 .....");
stmt.addBatch(" 5 .....");

try
{
    rows = stmt.executeBatch( );
}
catch(BatchUpdateException bue)
{
    int[ ] success_rows;
    success_rows = bue.getUpadteCounts( );
    System.err.println("Success Counts: " + success_rows );
}
```

不过,假若今天我们需要的是当全部的批处理都正确执行时,才真正交由数据库系统来执行,因此,我们必须使用交易(transaction)功能来达到我们的需求,笔者将上述程序修改为:

```
con.setAutoCommit(false);

Statement stmt = con.createStatement( );
int[ ] rows;
stmt.addBatch(" 1 .....");
stmt.addBatch(" 2 .....");
stmt.addBatch(" 3 .....");
stmt.addBatch(" 4 .....");
stmt.addBatch(" 5 .....");

rows = stmt.executeBatch( );
con.commit( );
```

上述范例中,先将 con 连接的自动认可模式(auto-commit)关闭,即 con.setAutoCommit(false),因此,惟有所有的批处理都没有任何错误发生时执行 con.commit(),程序才会执行批处理,即 stmt.executeBatch()。

14-7-5 BLOB 和 CLOB

JDBC 2.0 引进两个最重要的数据类型: BLOB 和 CLOB。BLOB 指的是二进制大型对象(Binary Large Object), CLOB 指的是字符大型对象(Character Large Object)。它们都是用来处理大量数据的数据类型, BLOB 和 CLOB 分别代表大量的二进制数据和文字数据。

或许有读者会问, SQL 92 不是已经提供 VARCHAR 和 VARBINARY 这两种数据类型吗? 为什么还需要 BLOB 和 CLOB 呢? 原因在于, VARCHAR 和 VARBINARY 在处理大量的数据时, 会产生两个严重的问题:

(1) VARCHAR 和 VARBINARY 当初被设计时, 只是用来储存较小量的数据, 并且它们所拥有的数据容量大小也有一定的限制, 因此当遇到数据容量大于它们的负荷时, 则会有错误发生, 以致无法正常运作。

(2) 当我们要从数据库取得 VARCHAR 和 VARBINARY 类型的数据时, 必须一次全部读出来。假若字段的数据内容超过 1MB, 执行查询指令时, 则查询结果将要等到每笔数据行都读取完毕, 这样的执行性能是十分低落的。当遇到这种情形时, 最好是使用串流(Streaming)来传送数据, 串流的方式可以直接抽取某些你真正需要的数据。因此使用串流会使得性能大为提升, 而 BLOB 和 CLOB 类型的数据则可以使用串流的方式来运作。

JDBC 2.0 支持两种 Java 类型, 以对应 SQL 的 BLOB 和 CLOB 类型: java.sql.Blob 和 java.sql.Clob。如果要从查询结果中取出数据, 方式就和其他的数据类型一样, 也是使用 getBlob() 和 getClob()。

当调用 getBlob() 和 getClob() 时, 你会取得一个 Blob 和 Clob 类型的对象, 它并不包含数据库中的任何数据, 这一点和其他 Java 数据类型不同。假若要用串流方式存取实际数据, 则可以这样做:

```
Blob b = rs.getBlob(1);
InputStream bin = b.getBinaryStream( );

Clob c = rs.getClob(2);
Reader charstr = c.getCharacterStream( );
```

我们还可以分块去取得真正需要的数据:

```
Blob b = rs.getBlob(1);
byte[ ] data = b.getBytes( 0 , b.length( ) );

Clob c = rs.getClob(2);
String text = c.getSubString( 0 , c.length( ) );
```

至于 BLOB 和 CLOB 的储存方式, 虽然 JDBC 2.0 并没有提供用来构建 BLOB 和 CLOB 实体的接口, 但是我们可以使用以下方式:

- 二进制数据: setBinaryStream() 或 setObject()
- 字符数据: setAsciiStream()、setUnicodeStream() 或 setObject()

14-8 JNDI - 数据来源(Data Source)与连接池(Connection Pool)

■ 数据来源(Data Source)

JDBC 2.0 提供 DataSource 的接口用于：减少你将一些数据库连接信息写死(hard code)在你的 JSP 程序代码之内。这样有什么好处呢？举个例子来说：假设有一天你想更换数据库，如果你采用写在 JSP 网页的方式，那么你将要一页一页修改 JSP 网页。若你在当初开发时，采用取得数据来源的方式来获得 Connection，那只要在 Server 上更改一下数据来源的设定就可以轻松完成工作。另外，使用数据来源最大的好处在于：Connection 的控管交由 Container 去控制，我们不用烦恼 Connection 管理上的问题。

■ 连接池(Connection Pool)

一般来说，如果每次 JSP 接收到请求时，就向数据库要求一个连接，当执行完就通知数据库中断连接，这样的方式将会耗费大量的时间与资源。因为数据库每次建立 Connection 时，都要先将 Connection 加载内存，再来验证用户的名称与密码，等到一切通过之后，再与用户建立连接，断线后又重新来一次。如此冗长的程序，既耗时且没有效率，因此我们将利用连接池的方式来解决这项问题。

连接池(Connection Pool)的运作方式是一开始向数据库要求很多的 Connection 储存在一个 Pool(池)内，让需要的人从连接池中取得 Connection，等到用完后再放回连接池内，其实说穿了就等于做一个缓冲区，让 JSP 与数据库之间能够获得最大的执行效率。

上面所提到的数据来源就是负责提供连接池的功用，而我们所需要做的就是从数据来源中的连接池取得 Connection，至于 Connection 的管理则交由数据来源处理。不过为了养成良好的习惯，还是将 JSP 不再使用的 Connection 对象归还给数据来源，让真正有需要的人来使用 Connection 对象。

■ JNDI

JNDI 全名为 Java Naming and Directory Interface。JNDI 主要提供应用程序所需资源上命名与目录的服务。在 J2EE 的环境中，JNDI 扮演了一个很重要的角色，它提供了一个接口让用户在不知道资源所在位置的情形下，取得该资源服务。

举个例子来说明，相信大家都用过网络磁盘驱动器的功能，如果有人事先将另一台机器上的磁盘驱动器接到用户的机器上，用户在使用机器的时候分辨不出现在的磁盘驱动器是存在本地端还是另一端的机器上，用户只须取得资源来使用，根本无须要知道资源位在何处。

JNDI 这个接口的底层基本上是 LDAP，LDAP 全名为 Lightweight Directory Access Protocol，讨论 LDAP 的内容超出本书的范围，故不在此叙述。读者只须知道 Server 将资源存放在类似

JSP2.0 技术手册

LDAP 上, 当用户须要取用 Server 相关资源的时候, 通过 JNDI 来取得 Server 资源即可。因此, 当我们要取得 DataSource 对象时, 所需要依靠的便是 JNDI。

在了解数据来源与 JNDI 之后, 笔者使用 Tomcat 5 来实现一个范例程序。首先我们必须设定好 JNDI 的 JDBC 数据来源。

■ 设定 JNDI 的 JDBC 数据来源和 DBCP 连接池

设定 Tomcat 的 JDBC 数据来源的步骤如下:

一、安装 JDBC Driver

将 JDBC Driver – *mysql-connector-java-3.0.9-stable-bin.jar* 放至 {Tomcat_Install}\common\lib 目录下。

二、设定 Tomcat 的 server.xml

我们必须修改 Tomcat 的 *server.xml*, 这个文件夹位于 {Tomcat_Install}\conf\server.xml。在 *server.xml* 中, 新增以下的设定值:

■ server.xml

```
<Server>
<Service>
<Engine>
<Host>
.....
<Context path="/JSPBook" docBase="JSPBook" debug="0" crosscontext=
    "true" reloadable="true">
<Resource name="jdbc/sample_db" auth="Container" type=
    "javax.sql.DataSource"/>
  <ResourceParams name="jdbc/sample_db">
    <parameter>
      <name>factory</name>
      <value>org.apache.commons.dbcp.BasicDataSourceFactory</value>
    </parameter>
    <parameter>
      <name>username</name>
      <value>root</value>
    </parameter>
    <parameter>
      <name>password</name>
      <value>browser</value>
    </parameter>
    <parameter>
      <name>driverClassName</name>
      <value>com.mysql.jdbc.Driver</value>
    </parameter>
    <parameter>
      <name>url</name>
```

JSP2.0 技术手册

```
<value>jdbc:mysql://localhost:3306/sample_db?useUnicode=
    true&characterEncoding=UTF-8</value>
</parameter>

<!-- Maximum number of dB connections in pool. Make sure you
    configure your mysqld max_connections large enough to handle
    all of your db connections. Set to 0 for no limit. -->
<parameter>
    <name>maxActive</name>
    <value>20</value>
</parameter>

<!-- Maximum number of idle dB connections to retain in pool.
    Set to 0 for no limit. -->
<parameter>
    <name>maxIdle</name>
    <value>5</value>
</parameter>

<!-- Maximum time to wait for a dB connection to become available
    in ms, in this example 10 seconds. An Exception is thrown if
    this timeout is exceeded. Set to -1 to wait indefinitely.
-->
<parameter>
    <name>maxWait</name>
    <value>10000</value>
</parameter>
</ResourceParams>
</Context>
....
<Host>
<Engine>
<Service>
<Server>
```

上述设定所代表的意思是在 JSPBook 站台中，定义一个 JDBC 数据来源，名称为 `jdbc/sample_db`。

补充

如果读者自己懒得设定，可以直接使用 JSPBook/CH14 的 config.txt，不过复制到 server.xml 时，必须小心贴对地方。

再来设定这个 `jdbc/sample_db` 的数据来源和 DBCP 连接池。如下：

```
<parameter>
    <name>factory</name>
    <value>org.apache.commons.dbcp.BasicDataSourceFactory</value>
</parameter>
```

设定使用 DBCP 连接池，这是由 Jakarta Project 组织所制作的连接池程序，它一样也是开放源代码的。

JSP2.0 技术手册

接下来是一般的 Mysql JDBC 设定, 其中包含: username、password、driverClassName 和 url, 如下所示:

```
<parameter>
  <name>username</name>
  <value>root</value>
</parameter>
<parameter>
  <name>password</name>
  <value>browser</value>
</parameter>
<parameter>
  <name>driverClassName</name>
  <value>com.mysql.jdbc.Driver</value>
</parameter>
<parameter>
  <name>url</name>
  <value>jdbc:mysql://localhost:3306/sample_db?useUnicode=true&
    characterEncoding=UTF-8</value>
</parameter>
```

这部分只有一个地方要注意: 设定 url 时, 为了解决中文问题, 须要加上 useUnicode 和 characterEncoding 两个参数。之前在 JSP 网页中, 两个参数只须用 & 符号相接即可。但是在 XML 文件中, 因为 & 是特殊符号, 所以不能直接使用 &, 必须改用 &

最后设定连接池的部分, 如下所示:

```
<!-- Maximum number of dB connections in pool. Make sure you
  configure your mysqld max_connections large enough to handle
  all of your db connections. Set to 0 for no limit. -->
<parameter>
  <name>maxActive</name>
  <value>20</value>
</parameter>

<!-- Maximum number of idle dB connections to retain in pool. Set to 0 for
  no limit. -->
<parameter>
  <name>maxIdle</name>
  <value>5</value>
</parameter>

<!-- Maximum time to wait for a dB connection to become available
  in ms, in this example 10 seconds. An Exception is thrown if
  this timeout is exceeded. Set to -1 to wait indefinitely. -->
<parameter>
  <name>maxWait</name>
  <value>10000</value>
</parameter>
```

maxActive 设定连接池中最多能有几个 Connection, 若为 0 时, 表示不限制。

maxIdle 设定连接池中最小能有几个 Connection, 若为 0 时, 表示不限制。

maxWait 设定一个 Connection 在执行时最长的闲置时间，单位为 ms。

三、设定 JSPBook 的 web.xml

设定 JSPBook 的 *web.xml*，如下：

```
<resource-ref>
  <description>JNDI JDBC DataSource of JSPBook</description>
  <res-ref-name>jdbc/sample_db</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

四、使用 JDBC 数据来源取得 Connection 对象

```
Context initContext = new InitialContext();
Context envContext = (Context)initContext.lookup("java:/comp/env");
DataSource ds = (DataSource)envContext.lookup("jdbc/sample_db");
con = ds.getConnection();
```

笔者在这里写一个范例：*Mysql_jndi.html* 和 *Mysql_jndi.jsp*。其中 *Mysql_jndi.jsp* 只将原本使用 JDBC 的方法连接数据库，改为使用 JNDI 的方式，先取得 JDBC 的数据来源，然后取得 Connection 对象。有兴趣的读者可以自行参阅那两个范例程序，在此笔者不再赘述。

14-9 JSTL 的 SQL 标签库

笔者已经在 7-4 节介绍过 SQL 标签库的使用。本节主要将之前的 *Mysql.jsp* 程序改用 JSTL 的写法，让读者更了解 SQL 标签库的使用。本章的范例程序一样有两个程序：*Mysql_jstl.html* 和 *Mysql_jstl.jsp*。

■ *Mysql_jstl.html*

```
<html>
<head>
  <title>CH14 - Mysql_jstl.html</title>
  <meta http-equiv="Content-Type" content="text/html; charset=GB2312">
</head>
<body>

<h2>将信息存入 Mysql 中 - 使用 JSTL 写法</h2>
<form name="form" action="Mysql_jstl.jsp" method="post" >
  <p>姓: <input name="last_name" type="text" id="last_name"></p>
  <p>名: <input name="first_name" type="text" id="first_name"></p>
  <p>
    <input type="submit" value="传送">
    <input type="reset" value="取消">
  </p>
</form>
```

JSP2.0 技术手册

```
</body>
</html>
```

Mysql_jstl.html 和 *Mysql.html* 一样, 只有 action 的对象有差异, 一个是 *Mysql_jstl.jsp*, 另一个是 *Mysql.jsp*。

■ *Mysql_jstl.jsp*

```
<%@ page contentType="text/html; charset=GB2312" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>

<html>
<head>
  <title>CH14 - Mysql_jstl.jsp</title>
</head>
<body>

  <h2>将信息存入 Mysql 中 - 使用 JSTL 写法</h2>

  <fmt:requestEncoding value="GB2312" />

  <c:set var="birth" value="1978/12/11" />
  <c:set var="sex" value="F" />
  <c:set var="email" value="aaa@asdf.com" />

  <sql:setDataSource driver="com.mysql.jdbc.Driver"
    url="jdbc:mysql://localhost:3306/sample_db?useUnicode=true&
    characterEncoding=UTF-8"
    user="root"
    password="browser" />

  <sql:update>
    INSERT INTO employee(employee_id, last_name, first_name, birth, sex,
      email)
    VALUES ( ? , ? , ? , ? , ? , ? )

    <sql:param value="${employee_id}" />
    <sql:param value="${param.last_name}" />
    <sql:param value="${param.first_name}" />
    <sql:param value="${birth}" />
    <sql:param value="${sex}" />
    <sql:param value="${email}" />
  </sql:update>

  <sql:query var="result">
    SELECT * FROM employee
  </sql:query>
```

```

从 employee 取出所有新增的姓名: <br>
<c:forEach items="${result.rows}" var="row" >
  新增姓名: ${row.last_name}
             ${row.first_name}<br>
</c:forEach>
</body>
</html>

```

不知读者有没有发现, *Mysql_jstl.jsp* 的程序代码比 *Mysql.jsp* 简洁。其中 JSTL 的 SQL 标签库是利用 `<sql:setDataSource>` 来与数据库取得联机的, 如下:

```

<sql:setDataSource driver="com.mysql.jdbc.Driver"
                  url="jdbc:mysql://localhost:3306/sample_db?useUnicode
                      =true& characterEncoding=UTF-8"
                  user="root"
                  password="browser" />

```

假若我们设定 JDBC 的数据来源, 则上述的程序代码可以更简洁:

```

<sql:setDataSource dataSource="jdbc/sample_db" />

```

只要这一行程序, 就可以取得与数据库的联机。接下来就是新增数据的部分:

```

<sql:update>
  INSERT INTO employee(employee_id, last_name, first_name, birth, sex,
                        email)
  VALUES ( ? , ? , ? , ? , ? , ? )

  <sql:param value="${employee_id}" />
  <sql:param value="${param.last_name}" />
  <sql:param value="${param.first_name}" />
  <sql:param value="${birth}" />
  <sql:param value="${sex}" />
  <sql:param value="${email}" />
</sql:update>

```

`<sql:update>` 可以用来新增、修改或删除数据。上述范例是使用 `PreparedStatement` 的方式来新增数据的工作, 其中新增数据的 SQL 语句有六个问号, 分别对应 `employee` 的六个字段。然后, 我们使用 `<sql:param>` 来设定这六个问号所代表的值。

最后使用 `<sql:query>` 从数据库取出我们想要的数据库, 如下:

```

<sql:query var="result">
  SELECT * FROM employee
</sql:query>

```

```

从 employee 取出所有新增的姓名: <br>
<c:forEach items="${result.rows}" var="row" >
  新增姓名: ${row.last_name}
             ${row.first_name}<br>
</c:forEach>

```

上述的范例也可以改为:

JSP2.0 技术手册

..... 略

从 employee 取出所有新增的姓名:


```
<c:forEach items="${result.rowsByIndex}" var="row" >
  新增姓名: ${row.[1]}
             ${row.[2]}<br>
</c:forEach>
```

Mysql_jstl.jsp 的执行结果如图 14-17 所示:

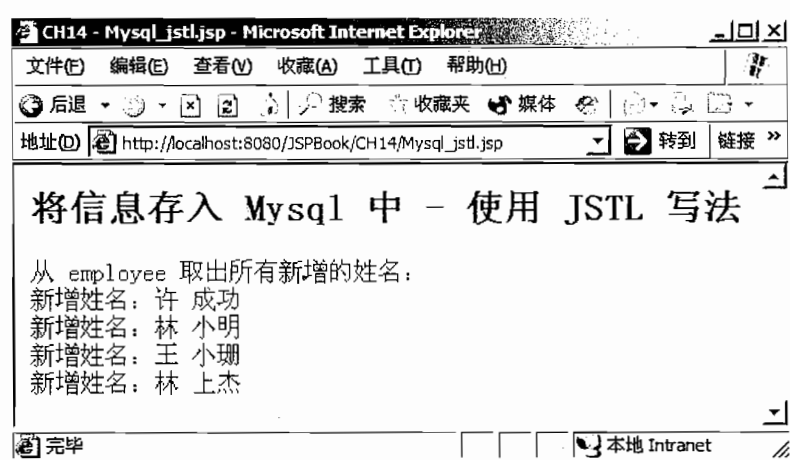


图 14-17 Mysql_jstl.jsp 的执行结果

注意

Mysql_jstl.jsp 是取出 employee 所有的数据, 因此, 假若你取出的数据有乱码时, 请先确认该笔数据是否为执行 Mysql.jsp 时所存入的数据。若是, 那是正常现象, 因为 Mysql.jsp 在储存数据时, 是使用 GB2312 的编码方式:

```
xxx/sample_db?user=root&password=xxx&useUnicode=true&
characterEncoding=GB2312"
```

而之后的范例, 都是使用 UTF-8 的编码方式, 所以才会造成这样的情形发生。结论就是 Mysql 的编码都使用 UTF-8 即可。

14-10 Connection Pool - Proxool

在前面章节中曾经介绍过 Tomcat 内附的连接池——DBCP。DBCP 是 Apache Jakarta 计划中的 Commons 成员之一, 因此 DBCP 也被集成进 Tomcat 之中, 只要用户简单地设定, 就能让 web 站台有连接池的功能。

除了 DBCP 之外, 笔者将介绍功能更强大、好用的连接池工具 Proxool。Proxool 和 DBCP 也是开放源代码, 读者可以至 <http://proxool.sourceforge.net/download.html> 下载最新的 binary 和 source

JSP2.0 技术手册

文件。图 14-18 为 Proxool 的首页。

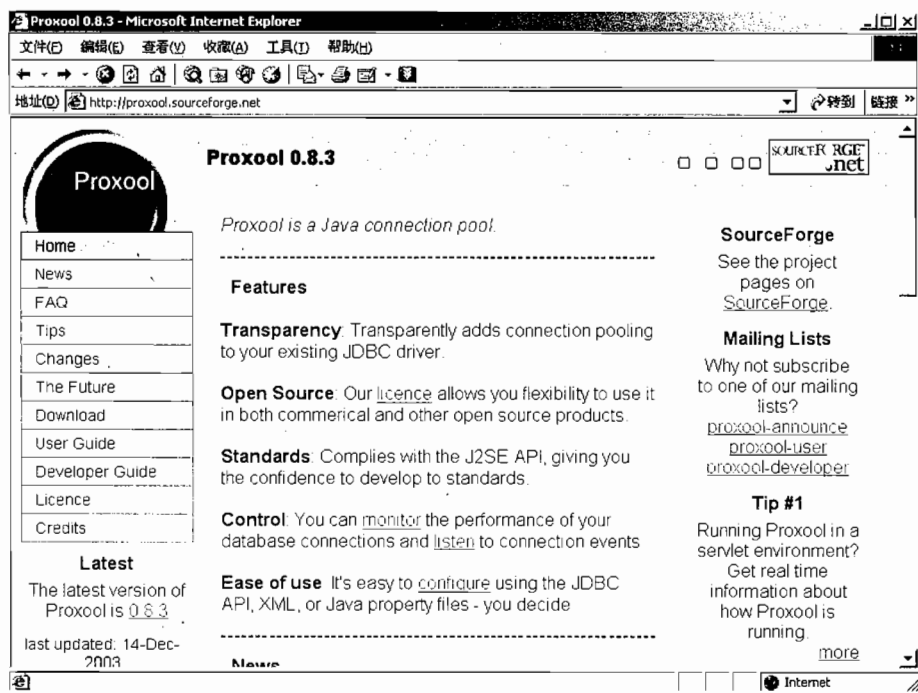


图 14-18 Proxool 首页(<http://proxool.sourceforge.net>)

14-10-1 安装与使用 Proxool

目前 Proxool 最新版本为 0.8.3, 读者可以自行至上述的网址下载最新的版本; 或者可以直接使用本书附赠光盘中的 *proxool-0.8.3.zip*。

proxool-0.8.3.zip 解压缩后, 将 *proxool-0.8.3/lib* 目录下的 *proxool-0.8.3.jar* 复制到站台的 *WEB-INF/lib* 之下。接下来改写之前 *Mysql.jsp* 的范例:

■ Proxool.jsp

```
<%@ page import="java.sql.*" %>
<%@ page contentType="text/html; charset=GB2312" %>

<html>
<head>
  <title>CH14 - Proxool.jsp</title>
</head>
<body>

<h2>使用 Proxool Connection Pool</h2>

<%
```

JSP2.0 技术手册

```

Connection con = null;
Statement stmt = null;
ResultSet rs = null;

String new_last_name = "";
String new_first_name = "";

try
{
    Class.forName("org.logicalcobwebs.proxool.ProxoolDriver");
    con = DriverManager.getConnection("proxool.JSPBook:
        com.mysql.jdbc.Driver:jdbc:mysql://localhost:3306/
        sample_db?user=root&password=browser&useUnicode=
        true&characterEncoding=UTF-8");
    stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
        ResultSet.CONCUR_UPDATABLE);
    String query = "SELECT * FROM employee";
    rs = stmt.executeQuery(query);

    while(rs.next()) {
        new_last_name = rs.getString("last_name");
        new_first_name = rs.getString("first_name");

        从 employee 取出姓名<%= new_last_name+new_first_name %><br>

    }
    stmt.close();
    con.close();
}
catch(SQLException sqle)
{
    out.println("sqle="+sqle);
}
finally
{
    try {
        if(con != null)
        {
            con.close();
        }
    }
    catch(SQLException sqle)
    {
        out.println("sqle="+sqle);
    }
}
}

```

Proxool.jsp 主要通过 Proxool 连接池取得 Connection, 然后显示 employee 表格中的

JSP2.0 技术手册

last_name 和 first_name。Proxool.jsp 最重要的程序代码如下：

```
..... 略
Class.forName("org.logicalcobwebs.proxool.ProxoolDriver");
con = DriverManager.getConnection("proxool.JSPBook:com.mysql.jdbc.Driver:
    jdbc:mysql://localhost:3306/sample_db?user=root&password=browser
    &useUnicode=true&characterEncoding=UTF-8");
stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
    ResultSet.CONCUR_UPDATABLE);
String query = "SELECT * FROM employee";
rs = stmt.executeQuery(query);
..... 略
```

首先动态加载 Proxool 的 driver:

```
Class.forName("org.logicalcobwebs.proxool.ProxoolDriver");
```

然后将 Proxool URL 分为三个部分：连接池的别名、原本 JDBC 驱动程序的名称、原本 URL 的名称，如下所示：

```
proxool.JSPBook:com.mysql.jdbc.Driver:jdbc:mysql://localhost:3306/sample_db?
user=root&password=browser&useUnicode=true&characterEncoding=UTF-8
```

其中 proxool.JSPBook 的 JSPBook 就是连接池的别名；com.mysql.jdbc.Driver 为 JDBC 驱动程序的名称；jdbc:mysql://localhost:3306/sample_db? user=root&password= browser &useUnicode=true&characterEncoding=UTF-8 则为原本的 URL 设定，它们三者之间都用冒号(:)来做分隔。

Proxool.jsp 的执行结果如图 14-19 所示：

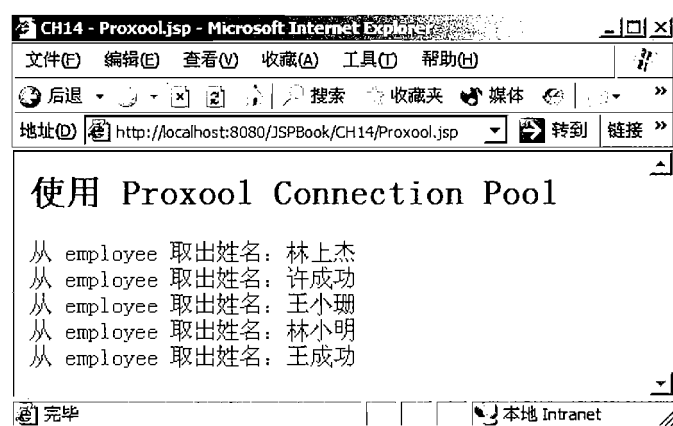


图 14-19 Proxool.jsp 的执行结果

14-10-2 设定 Proxool

Proxool 提供许多连接池的设定参数，例如：连接池最多有几个 Connection、最少有几个

Connection、Connection 的生命期限等等。

Proxool 提供几种设定的方式:

(一) 通过 java.util.Properties 对象来设定, 例如:

```
Properties info = new Properties();
info.setProperty("proxool.maximum-connection-count", "20");
info.setProperty("proxool.house-keeping-test-sql", "select
    CURRENT_DATE");
info.setProperty("user", "root");
info.setProperty("password", "browser");
String alias = "JSPBook";
String driverClass = "com.mysql.jdbc.Driver";
String driverUrl = "jdbc:mysql://localhost:3306/sample_db?
    useUnicode=true&characterEncoding=UTF-8";
String url = "proxool." + alias + ":" + driverClass + ":" + driverUrl;
connection = DriverManager.getConnection(url, info);
```

(二) 先通过 XML 文件来设定, 例如:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- the proxool configuration can be embedded within your own application's.
    Anything outside the "proxool" tag is ignored. -->
<proxool>
    <alias>JSPBook</alias>
    <driver-url>
        jdbc:mysql://localhost:3306/sample_db?useUnicode=true&characterEncoding=
            UTF-8
    </driver-url>
    <driver-class> com.mysql.jdbc.Driver</driver-class>
    <driver-properties>
        <property name="user" value="root"/>
        <property name="password" value="browser"/>
    </driver-properties>
    <maximum-connection-count>10</maximum-connection-count>
    <house-keeping-test-sql>select CURRENT_DATE</house-keeping-test-sql>
</proxool>
```

然后通过 JAXPConfigurator 读取 XML 文件:

```
JAXPConfigurator.configure("/WEB-INF/classes/proxool.xml", false);
```

(三) 先通过 Properties 文件来设定, 例如:

```
jdbc-0.proxool.alias=JSPBook
jdbc-0.proxool.driver-url= jdbc:mysql://localhost:3306/
    sample_db?useUnicode=true&characterEncoding=UTF-8
jdbc-0.proxool.driver-class=com.mysql.jdbc.Driver
jdbc-0.user=root
jdbc-0.password=browser
jdbc-0.proxool.maximum-connection-count=10
jdbc-0.proxool.house-keeping-test-sql=select CURRENT_DATE
```

然后通过 PropertyConfigurator 读取 Properties 文件:


```
PropertyConfigurator.configure("/WEB-INF/classes/Proxool.properties");
```

(四) 在 *web.xml* 中, 通过 Servlet 来设定。方法又分为三种, 前两种是依设定文件格式而定:

1. XML 文件

```
<servlet>
  <servlet-name>ServletConfigurator</servlet-name>
  <servlet-class>
    org.logicalcobwebs.proxool.configuration.ServletConfigurator
  </servlet-class>
  <init-param>
    <param-name>xmlFile</param-name>
    <param-value>WEB-INF/classes/Proxool.xml</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
```

2. Properties 文件

```
<servlet>
  <servlet-name>ServletConfigurator</servlet-name>
  <servlet-class>
    org.logicalcobwebs.proxool.configuration.ServletConfigurator
  </servlet-class>
  <init-param>
    <param-name>propertyFile</param-name>
    <param-value>WEB-INF/classes/Proxool.properties</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
```

3. Init Parameter

```
<servlet>
  <servlet-name>ServletConfigurator</servlet-name>
  <servlet-class>
    org.logicalcobwebs.proxool.configuration.ServletConfigurator
  </servlet-class>
  <init-param>
    <param-name>jdbc-0.proxool.alias</param-name>
    <param-value>JSPBook</param-value>
  </init-param>
  <init-param>
    <param-name>jdbc-0.proxool.driver-url</param-name>
    <param-value>
      jdbc:mysql://localhost:3306/sample_db?user=root&password=browser&
      useUnicode=true&characterEncoding=UTF-8
    </param-value>
  </init-param>
  <init-param>
    <param-name>jdbc-0.proxool.driver-class</param-name>
    <param-value>com.mysql.jdbc.Driver</param-value>
  </init-param>
```



```
</servlet>
```

笔者采用的是第四种方式，好处在于当 Container 启动时，Proxool 的参数会自动设定加载内存中，原因在于：

```
<servlet>
..... 略
  <load-on-startup>1</load-on-startup>
</servlet>
```

完整的范例如下：

■ web.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
  version="2.4">
..... 略
  <servlet>
    <servlet-name>ServletConfigurator</servlet-name>
    <servlet-class>
      org.logicalcobwebs.proxool.configuration.ServletConfigurator
    </servlet-class>
    <init-param>
      <param-name>propertyFile</param-name>
      <param-value>WEB-INF/classes/Proxool.properties</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
..... 略
</web-app>
```

■ Proxool.properties

```
jdbc-0.proxool.alias=JSPBook
jdbc-0.proxool.driver-class=com.mysql.jdbc.Driver
jdbc-0.proxool.driver-url=jdbc:mysql://localhost:3306/sample_db?user=
  root&password=browser
  &useUnicode=true&characterEncoding=UTF-8

jdbc-0.proxool.maximum-connection-count=10
jdbc-0.proxool.prototype-count=4

jdbc-0.proxool.house-keeping-test-sql=select CURRENT_DATE
jdbc-0.proxool.verbose=true
```

■ Proxool-config.jsp

```
<%@ page import="java.sql.*" %>
<%@ page contentType="text/html; charset=GB2312" %>

<html>
```

JSP2.0 技术手册

```
<head>
  <title>CH14 - Proxool-config.jsp</title>
</head>
<body>

<h2>使用 Proxool Connection Pool</h2>

<%
  Connection con = null;
  Statement stmt = null;
  ResultSet rs = null;

  String new_last_name = "";
  String new_first_name = "";

  try
  {
    con = DriverManager.getConnection("proxool.JSPBook");
    stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                               ResultSet.CONCUR_UPDATABLE);
    String query = "SELECT * FROM employee";
    rs = stmt.executeQuery(query);

    while(rs.next()) {
      new_last_name = rs.getString("last_name");
      new_first_name = rs.getString("first_name");
%>
      从 employee 取出姓名: <%= new_last_name+new_first_name %><br>
<%
    }

    stmt.close();
    con.close();
  }
  catch(SQLException sqle)
  {
    out.println("sqle="+sqle);
  }
  finally
  {
    try {
      if(con != null)
      {
        con.close();
      }
    }
    catch(SQLException sqle)
    {
      out.println("sqle="+sqle);
    }
  }
%>
```

JSP2.0 技术手册

```
</body>
</html>
```

Proxool-config.jsp 和 *Proxool.jsp* 的区别在于取得 Connection 的程序部分, 原本 *Proxool.jsp* 为:

```
Class.forName("org.logicalcobwebs.proxool.ProxoolDriver");
con = DriverManager.getConnection("proxool.JSPBook:com.mysql.jdbc.Driver:
    jdbc:mysql://localhost:3306/sample_db?user=root&password=browser
    &useUnicode=true&characterEncoding=UTF-8");
```

Proxool-config.jsp 因为已经设定 ServletConfigurator, 所以程序代码改为:

```
con = DriverManager.getConnection("proxool.JSPBook");
```

只需要一行即可顺利取得 Connection 对象。*Proxool-config.jsp* 的程序代码是不是变得简洁许多。

补充

有关 Proxool 的参数设定值, 可以参考下列网址:

<http://proxool.sourceforge.net/properties.html>

14-10-3 Proxool 后端统计接口

Proxool 另一个特点就是: Proxool 有一个后端统计接口。使用的方法十分简单, 只需要在 *web.xml* 中设定好 AdminServlet 即可使用。

■ web.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
    version="2.4">
    ..... 略
    <servlet>
        <servlet-name>Admin</servlet-name>
        <servlet-class>org.logicalcobwebs.proxool.admin.servlet.AdminServlet
        </servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>Admin</servlet-name>
        <url-pattern>/Admin</url-pattern>
    </servlet-mapping>
    ..... 略
</web-app>
```

设定好之后, 再来新增 Proxool 的参数 `jdbco.proxool.statistics`。因此笔者再修改 *Proxool.properties* 文件:

JSP2.0 技术手册

■ Proxool.properties

略

```
jdbc-0.proxool.statistics=10s,1m,1d
```

以 JSPBook 站台为例, 输入:

<http://localhost:8080/JSPBook/Admin>

即可看到如图 14-20 所示的 Proxool Connection Pool 统计接口。

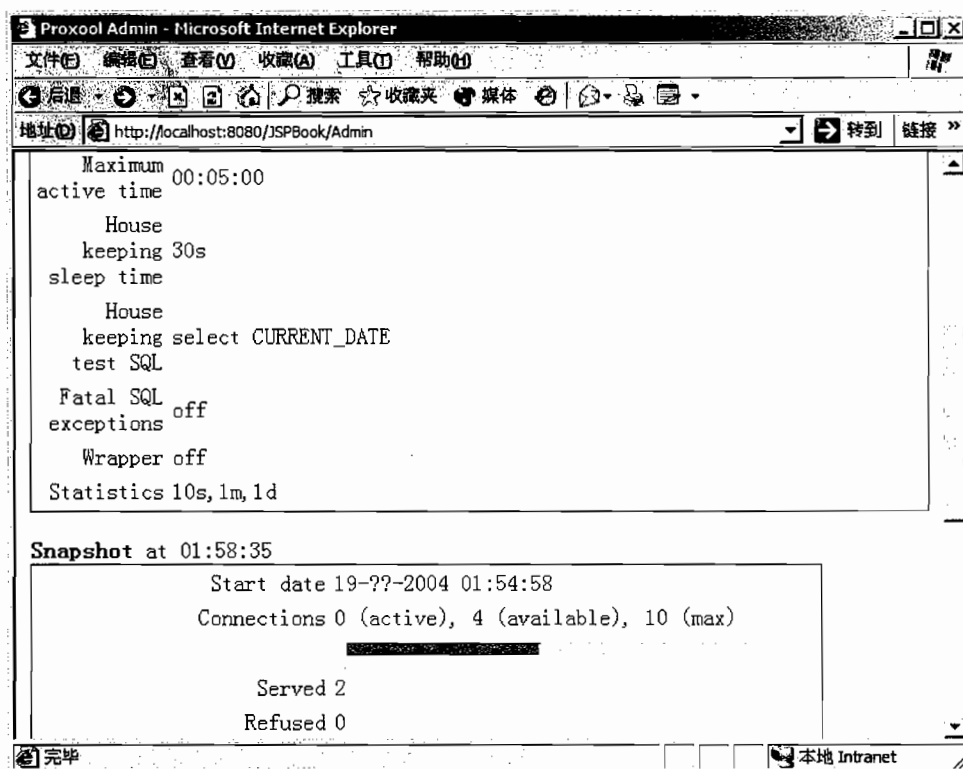


图 14-20 Proxool Connection Pool 统计接口

15

第十五章

JSP Tag Library

本章将为读者介绍 JSP 标签函数库(Tag Library)的制作和使用。所谓 JSP 标签函数库是指提供给开发人员能够自定义标签的功能，加以重复利用。本章将分 4 节来介绍：

- 15-1 JSP Tag Library 简介
- 15-2 一个简单的 Tag Library 范例
- 15-3 Tag Handler Class
- 15-4 Tag Library 范例程序

JSP2.0 技术手册

15-1 JSP Tag Library 简介

在 JSP 1.1 时, 新增 JSP 标签函数库技术, 它最大的功用在于让用户能够自行制订一个标签, 例如: 我们可以写一个寄 E-Mail 的标签, 或是专门为数据库处理、运算的标签。如下所示:

```
<%@ taglib uri = "/myTab" prefix="mytag" %>
<mytag:mail host="mailserver" sender="Tim@b.com" recipient="Mike@a.com">
Hello ...
</mytag:mail>
```

或者

```
<%@ taglib uri = "/myTab" prefix="mytag" %>
<%
    Class.forName("...").getInstance( );
    Connection con = DriverManager.getConnection("...");
%>
<mytag:sql connection="<%= con %>" user="browser" password="123">
Select * From Table
</mytag:sql>
```

15-1-1 标签函数库和 JavaBean 的比较

看了上述标签函数库的使用范例后, 读者会不会觉得使用上很方便, 不过一定会有读者认为标签函数库和 JavaBean 的功能好像很类似, 功能上是否会有重叠。不错, 当初笔者在钻研标签函数库时, 也觉得它和 JavaBean 一样, 都是将比较复杂的逻辑运算写到组件(指标签函数库和 JavaBean)之中, 这样一来, 除了未来维护能够省力、省时外, 并且还可以重复使用这些组件。但是制作标签比写一个 JavaBean 还要麻烦, 不过标签函数库有两项比 JavaBean 更具有优势:

- (1) 标签函数库能够方便处理网页内容的数据;
- (2) 标签函数库相对于网页开发者而言, 在使用上比 JavaBean 更加容易上手。

读者看到第(1)项的优势, 或许不是很清楚, 不过看完后面的范例程序后, 相信您自然就能了解了。第(2)项的优势中, 因为标签函数库在使用上也和开发 HTML 一样使用标签语法, 因此对于网页开发者, 可能会比较熟悉且较容易上手。

15-1-2 标签函数库的运作

本书曾在“第四章: JSP 语法”当中介绍 taglib 这个指令, 它有两个属性: uri 和 prefix。uri 属性主要是告诉 Container, .tld 文件的位置; 而 prefix 属性主要想取一个别名来代表标签, 并且用来区分多个标签函数库。

注意

所谓 TLD, 就像标签的设定文件, 其中包括: 标签的名称、标签的简述、标签的 Handler Class 和标签用到的属性, 等等。

当 Container 编译自定义标签时, 首先会判断是否须要加载 *tld* 文件, 而 *tld* 就是 Tag Library Descriptor 的缩写。如果是第一次编译的标签, Container 则会根据 *uri* 的位置, 将 *tld* 文件加载至 JVM, 假若在编译另一个 JSP 网页, 遇到使用相同 *tld* 文件的标签时, *tld* 文件就不需要再重新加载至 Container 的 JVM 中。如图 15-1 所示。



图 15-1 标签函数库的运作

15-2 一个简单的 Tag Library 范例

上一节大略了解了标签函数库之后, 笔者将先写一个简单的范例程序 Hello, 让读者了解标签函数库制作的流程有哪几个步骤, 然后到下一节才开始详细解说标签函数库。因为笔者自己在钻研标签函数库时, 觉得看理论很难了解所要表达的内容, 所以笔者先从范例着手。

接下来开始介绍第一个标签函数库的范例程序 Hello World。通常在制作一个完整的标签函数库程序时, 需要三个组件:

1. 标签处理类 (Tag Handler Class)
2. 标签性质文件 (Tag Library Descriptor File)
3. JSP 网页

我们的第一个范例程序包含上述三个组件。不过, 这三个组件在下一节才会有更深入的介绍。

现在定义一个叫做<Hello>的标签, 当这个标签被调用时, 会在页面上显示出 “Hello World Use Tag Library ” 字符串。

15-2-1 Hello 标签的标签处理类

■ Hello.java

```

package tw.com.javaworld.CH15;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;

public class Hello extends TagSupport {

```

```
public int doStartTag() {
    try {
        JspWriter out = pageContext.getOut();
        out.println("Hello World Use Tag Library");
    } catch (Exception e) {
        System.out.println("Hello Tag Error : " + e);
    }
    return (SKIP_BODY);
}
```

当我们在定义一个新的标签时，首先要定义一个 Java 类，即所谓的标签处理类(Tag Handler Class)，它告诉 Container 遇到自定义标签时，要执行哪些工作。

```
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
```

首先，*Hello.java* 一开始(import)导入两个套件：`javax.servlet.jsp.tagext.*` 和 `javax.servlet.jsp.*`，这是每次在制作标签处理类时，一定要导入的套件。

```
public class Hello extends TagSupport { }
```

目前我们只是定义一个单纯显示文字的空元素(empty element)标签，如<Hello/>，因此，Hello 类要继承 TagSupport 类，下一节将为读者做更详细的介绍。

```
public int doStartTag( ) {
    try {
        JspWriter out = pageContext.getOut( );
        out.println("Hello World Use Tag Library");
    }
    catch (Exception e)
    {
        System.out.println("Hello Tag Error :"+e);
    }
    return(SKIP_BODY);
}
```

在 TagSupport 类中，定义好 doStartTag()这个方法。doStartTag()主要是指当 Container 遇到所定义的起始标签时，它会根据 doStartTag()方法中的内容做处理。

因为这个范例主要是显示字符串的 Hello 标签，于是我们首先要利用 `pageContext.getOut()` 来取得 JspWriter，然后才能将字符串显示至网页上。在第五章笔者曾介绍过 pageContext 对象，在此不多说明。

然后，程序执行完毕后，我们一定要回传一个值，告诉 Container 处理完起始标签后，下一步该如何？在这个范例中，我们回传一个 SKIP_BODY 的值，SKIP_BODY 的意思是告诉 Container，不需对本体内容(body content)做处理。

补充

所谓本体内容是指：在我们自定义的起始标签和结尾标签之间的内容，如下所示：

<起始标签>

.....

..... ← 本体内内容

.....

</结尾标签>

因为现在定义的 Hello 标签是属于空元素(empty element)标签, 因此根本没有所谓的本体内内容, 所以当然要回传 SKIP_BODY 的值。

最后, 把 *Hello.java* 编译后, 会产生 *tw.com.javaworld.CH15\Hello.class*。tw.com.javaworld.CH15 是套件的名称。现在假设 Tomcat 5.0 当做我们的 Container, 则我们可以将它们放至下列的目录:

```
{Tomcat_Install}\webapps\JSPBook\WEB-INF\classes
```

形成:

```
{Tomcat_Install}\webapps\JSPBook\WEB-INF\classes\tw\com\javaworld\CH15\Hello.class
```

现在已经完成标签函数库三个步骤中的第一步: 标签处理类; 接着就是第二步: 标签性质文件 (Tag Library Descriptor File)。

补充

编译标签处理类时, 必须将 jsp-api.jar 加入 CLASSPATH。jsp-api.jar 可以在 Tomcat 5.0.16 安装目录下的 common\lib\jsp-api.jar 找到。

15-2-2 Hello 标签的 Tag Library Descriptor File

当我们定义好 Tag Handler 之后, 下一个工作就是开始定义 Tag Library Descriptor, 简称 TLD。所谓的 TLD, 就像标签的设定文件, 其中包括: 标签的名称、标签的简述、标签的处理类和标签用到的属性, 等等。它最大的特色在于, 设定文件的格式是 XML 文件, 且设定的项目和功能是根据 Sun 公司所定义的。

补充

有关 TLD 文件的元素介绍, 请参阅 “16-4: Tag Library Descriptor”。

■ MyTaglib.tld

```
<?xml version="1.0" encoding="UTF-8" ?>
```

```
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd"
  version="2.0">

  <description>My Taglib by JavaWorld.com.tw</description>
  <tlib-version>1.0</tlib-version>
  <jsp-version>2.0</jsp-version>
  <short-name>Mytaglib</short-name>
```

JSP2.0 技术手册

```

    <uri></uri>

    <tag>
        <description>Simplest example:Hello</description>
        <name>Hello</name>
        <tag-class>tw.com.javaworld.CH15.Hello</tag-class>
        <body-content>empty</body-content>
    </tag>

</taglib>

```

在 *MyTaglib.tld* 中定义一组 tag，名称为 Hello，它的处理类是 *tw.com.javaworld.CH15.Hello*。设定好之后，我们将它放置在

```
{Tomcat_Install}\webapps\JSPBook\WEB-INF\tlds\
```

之下，形成

```
{Tomcat_Install}\webapps\JSPBook\WEB-INF\tlds\MyTaglib.tld
```

于是第二个步骤也已完成，然后就是最后一个步骤，利用 JSP 网页调用我们所定义好的标签。

15-2-3 JSP 网页

接下来，我们写一个 *Hello.jsp*，看看如何在 JSP 网页中调用所定义的标签。

■ *Hello.jsp*

```

<%@ page contentType="text/html;charset=GB2312" %>
<%@ taglib uri="/WEB-INF/tlds/MyTaglib.tld" prefix="mytag" %>

<html>
<head>
    <title>CH15 - Hello.jsp</title>
</head>
<body>

<h2>Hello World 标签</h2>

<h1><mytag:Hello /></h1>

</body>
</html>

```

在 JSP 网页中调用我们的自定义标签时，须要用到 *taglib* 指令，*taglib* 指令有两个参数须要设定：*uri* 和 *prefix*，所谓的 *uri* 就是 TLD 所在的位置。不过我们也可以修改 *web.xml* 中设定 *taglib* 的 *uri* 的对映(mapping)，我们在 15-2-4 小节中会有更加详细的说明。

prefix 最大的好处在于：当我们在 JSP 网页中用到很多个 TLD 所定义的标签，以及遇到有相同名称的标签时，可以用来区分到底使用哪一个 TLD 所定义的标签，例如：

```

<%@ taglib uri="/WEB-INF/MyTaglib1.tld" prefix="mytag1" %>
<%@ taglib uri="/WEB-INF/MyTaglib2.tld" prefix="mytag2" %>
<%@ taglib uri="/WEB-INF/MyTaglib3.tld" prefix="mytag3" %>

```

JSP2.0 技术手册

```

<mytag1:tag1>
JSP
</mytag1:tag1>
<mytag2:tag1>
</mytag2:tag1>
<mytag3:tag1>
</mytag3:tag1>

```

最后，*Hello.jsp* 的执行结果如图 15-2 所示：



图 15-2 <Hello>标签的执行结果

15-2-4 Tag Library & web.xml

我们用之前的 Hello Tag 为范例，假设先不设定 *web.xml* 的内容，将 *MyTaglib.tld* 文件放在：
{Tomcat_Install}\webapps\JSPBook\WEB-INF\tlds\

而我们的 JSP 网页就要这样表示：

```
<%@ taglib uri="/WEB-INF/tlds/MyTaglib.tld" prefix="mytag" %>
```

不过，这样的做法有两个缺点：

- (1) 每次都要输入很长的 uri，非常麻烦。
- (2) 假若 tld 变动位置时，则每个 JSP 网页也需要跟着修改，这样很没有效率。

因此，我在这里建议读者通过修改 *web.xml* 的内容，就能够轻松解决这些问题，例如：

■ web.xml

```

<taglib>
  <taglib-uri>Taglib</taglib-uri>
  <taglib-location>/WEB-INF/tlds/MyTaglib.tld</taglib-location>
</taglib>

```

此时，*Hello.jsp* 中 taglib 指令只须写成：

```
<%@ taglib uri="Taglib" prefix="mytag" %>
```


就能够顺利执行标签函数库。

其实我们可以把<taglib-uri>想成是帮 TLD(如: *MyTaglib.tld*)取一个别名, 如果在 JSP 网页中执行 taglib 指令时, 在参数 uri 中输入别名, Container 就会根据 *web.xml* 中的设定环境找到所对应的 TLD, 而这个对应就是利用<taglib>中的<taglib-uri>和<taglib-location>来完成的。

15-3 Tag Handler Class

读者看完上面 Hello 的范例程序后, 应该对自定义标签的流程和方法有了基本的了解。接下来的两节, 笔者将会针对标签处理类做更详细的说明。

所谓 Tag Handler Class, 中文译为标签处理类, 就是自定义标签所要处理的事务, 例如:<Hello>标签用来显示字符串、<Email>标签用来寄发 E-Mail、<Sql>标签用来连接数据库, 等等。当 Container 在编译 JSP 网页, 遇到用户自定义的标签时, Container 会先至 TLD(Tag Library Descriptor)找到自定义标签的标签处理类, 然后将数据传送至标签处理类做运算、处理, 因此我们也称标签处理类是标签函数库的核心部分。

不过, 当在说明标签处理类时, 其实也等于介绍了 JSP 2.0 中有关标签函数库的 API。因此, 笔者将为各位介绍 JSP 2.0 中所有和标签函数库有关的 API。

15-3-1 API 概观

读者由上一章的范例程序当中可以很清楚地知道, 在开发标签处理类时, 必须要导入 javax.servlet.jsp.tagext.* 和 javax.servlet.jsp.* 两个套件。所以接下来我们就先来说明这两个套件:

1. javax.servlet.jsp 套件

javax.servlet.jsp 套件有六个主要的类: ErrorData、JspContext、JspEngineInfo、JspFactory、JspWriter 和 PageContext。

ErrorData 类主要包含有关错误信息的数据。JspContext 是一个抽象类, 它提供给继承于它的类一些便利的方法, 例如: 管理所有范围的变量、使用 JspWriter 输出数据等等。

JspEngineInfo 是一个抽象类, 它能提供 Container 的相关信息。例如: 你可以利用 JspEngineInfo.getSpecificationVersion()来取得目前 Container 的版本。

JspWriter 和 PageContext 这两个类最常使用, 例如: 在网页上显示字符串时:

```
JspWriter out = pageContext.getOut();
```

或者取得请求的参数时:

```
ServletRequest request = pageContext.getRequest();  
String Parameter = request.getParameter("Para1");
```

JspWriter 所声明的对象类型和 out 隐含对象一样, 它提供下列几个常用的方法(见表 15-1)。

表 15-1

方 法	说 明
clear()	清除输出缓冲区的内容
clearBuffer()	清除输出缓冲区的内容
close()	关闭输出流, 清除所有的内容
getBufferSize()	取得目前缓冲区的大小(KB)
getRemaining()	取得目前使用后还剩下的缓冲区大小(KB)
newLine()	写入一换行的字符
print()	输出字符或是字符串
println()	输出字符或是字符串
isAutoFlush()	如果回传 true, 表示缓冲区满会自动清除; 若为 false, 表示缓冲区满了不会自动清除, 而会产生异常处理

PageContext 所声明的对象类型和 pageContext 隐含对象一样, 它提供下列几个常用的方法 (见表 15-2)。

表 15-2

方 法	说 明
getException()	回传目前网页的异常, 不过此网页要为 error page, 如: exception 隐含对象
getOut()	回传目前网页的输出流, 如: out 隐含对象
getPage()	回传目前网页的 Servlet 实体(instance), 如: page 隐含对象
getRequest()	回传目前网页的请求(request), 如: request 隐含对象
getResponse()	回传目前网页的响应(response), 如: response 隐含对象
getServletConfig()	回传目前此网页的 ServletConfig 对象, 如: config 隐含对象
getServletContext()	回传目前此网页的执行环境(context), 如: application 隐含对象
getSession()	回传和目前网页有联系的会话(session), 如: session 隐含对象
getAttribute(name, scope)	回传属性名称为 name, 范围为 scope 的属性对象, 回传类型为 java.lang.Object
getAttributeNamesInScope(scope)	回传所有属性范围为 scope 的属性名称, 回传类型为 Enumeration
getAttributeScope(name)	回传属性名称为 name 的属性范围
removeAttribute(name)	移除属性名称为 name 的属性对象
removeAttribute(name, scope)	移除属性名称为 name, 范围为 scope 的属性对象
setAttribute(name, value, scope)	指定属性对象的名称为 name、值为 value、范围为 scope
findAttribute(name)	寻找在所有范围中属性名称为 name 的属性对象

2. javax.servlet.jsp.tagext 套件

javax.servlet.jsp.tagext 包括八个接口, 笔者将它们分为三大类, 如下所示:

第一大类为 BodyTag、Tag、IterationTag

第二大类为 JspTag、SimpleTag、JspFragment

第三大类为 DynamicAttributes、TryCatchFinally

第一大类 BodyTag、Tag、IterationTag 是 JSP 2.0 之前就有的接口。本节主要介绍这三个。

第二大类 JspTag、SimpleTag、JspFragment 是 JSP 2.0 新增加进来的功能，主要用来简化自定义标签的实现过程。JSP 2.0 多了 Tag File、Simple Tag，这将在第十六章介绍。

第三大类 DynamicAttributes、TryCatchFinally。DynamicAttributes 是 JSP 2.0 新增的接口。当一个标签实现 DynamicAttributes 接口时，它可以轻易接收动态属性至标签中处理。

TryCatchFinally 接口主要用来辅助 BodyTag、Tag 和 IterationTag 处理异常事件，它提供两个方法 doCatch() 和 doFinally()。假若标签处理类实现 TryCatchFinally 接口，当处理自定义标签的本体内容时，产生异常事件或是标签处理类的方法，如：doStartTag()、doEndTag()、doAfterBody() 或 doInitBody() 抛出异常事件就会调用 doCatch() 方法，这和 Java 标准的 catch 一样。假若没有任何异常事件发生，标签处理类的 doEndTag() 方法被调用后，doFinally() 方法会立刻被调用执行。

javax.servlet.jsp.tagext 包括十七个类：

BodyContent	TagAttributeInfo	TagLibraryValidator
BodyTagSupport	TagData	TagSupport
FunctionInfo	TagExtraInfo	TagVariableInfo
PageData	TagFileInfo	ValidationMessage
SimpleTagSupport	TagInfo	VariableInfo
TagAdapter	TagLibraryInfo	

其中，TagSupport 和 BodyTagSupport 这两个类将是本节的主轴，但笔者也会介绍其他重要的类，让读者更能够了解标签函数库。在说明 TagSupport 和 BodyTagSupport 时，先给读者介绍一些基本概念。

一开始接触标签函数库的读者，也许会搞不清楚 TagSupport 和 BodyTagSupport 的区别。因此，我用最简单的方法来区分在开发标签处理类时，什么时候要继承 TagSupport 类，什么时候要继承 BodyTagSupport 类。以下是我的区分方式：

自定义标签时，如果将对本体内容做一些运算处理，则写的标签处理类必须继承 BodyTagSupport 类；反之，则继承 TagSupport 类。以下举例说明之：

假设我自定义一个名为<jsp2html>的标签，它的功能是直接在网页上显示 JSP 的程序代码，我必须将 <,>,” 和 & 替代为 “<”，“>”，“"” 和 “&”

```
<prefix:jsp2html>
<%
    .....
    .....
    .....
%>
</prefix:jsp2html>
```

因为我必须将<prefix:jsp2html>和</prefix:jsp2html>之间的内容全部“抓”到标签处理

JSP2.0 技术手册

类中，将一些特殊符号<,>," 和 &都转换为<, >, "和&, 因此<jsp2html>的标签处理类必须继承 BodyTagSupport 类。

15-3-2 TagSupport 类

TagSupport 类实现 Tag、IterationTag、JspTag 和 Serializable 接口，其中 Tag 和 IterationTag 接口是本节的重点。在 TagSupport 类中，主有几个方法最为重要，它们分别为：doStartTag()、doAfterBody()、doEndTag()和 release()，如下所示：

```
public int doStartTag() throws JspException  
public int doAfterBody() throws JspException  
public int doEndTag() throws JspException  
public void release()
```

前面提到的 TagSupport 类主要是实现 Tag 接口和 IterationTag 接口。实做 Tag 接口的标签，只有最基本处理的功能，Tag 接口主要的方法有 doStartTag()、doEndTag()和 release()。图 15-3 为实现 Tag 接口标签的流程图：

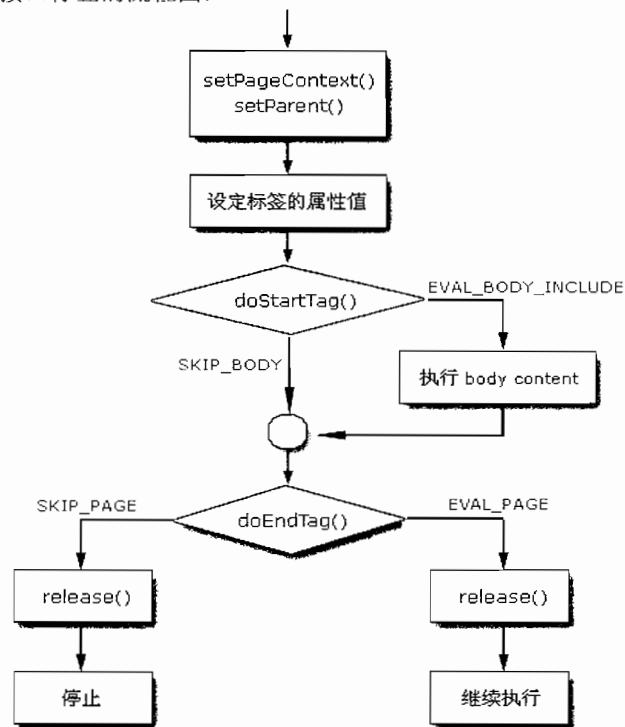


图 15-3 Tag 接口标签的生命周期

至于实现 IterationTag 接口的标签，除了和 Tag 接口一样的功能和方法外，它还增加了可重复处理本体内容的功能。这项功能是通过 doAfterBody()方法来达成的。图 15-4 为实现 IterationTag 接口标签的流程图：

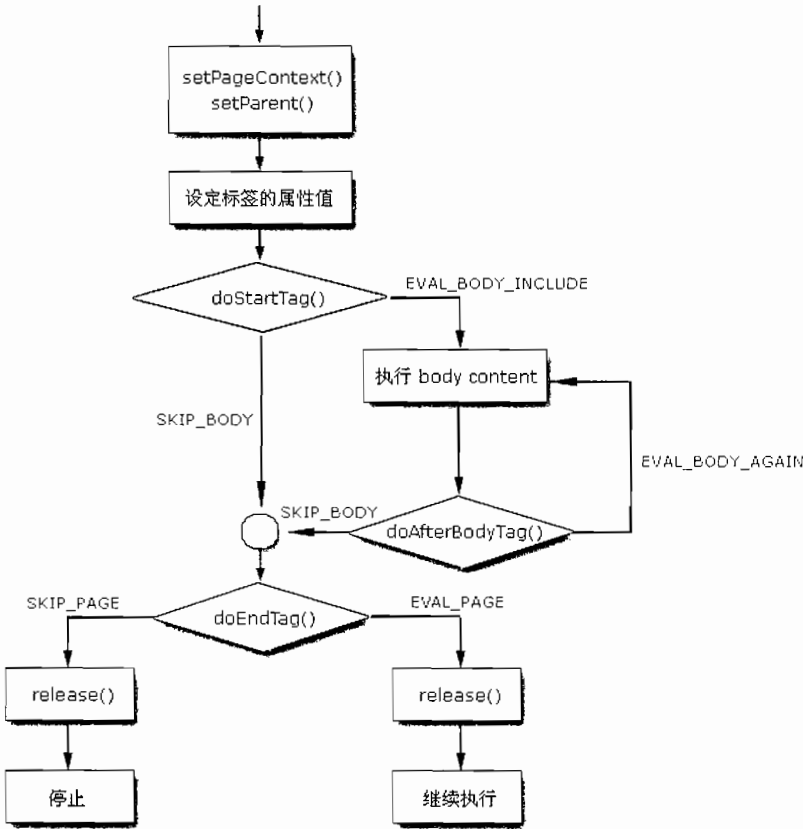


图 15-4 IterationTag 接口标签的生命周期

接下来笔者介绍 TagSupport 类中四个重要的方法：

■ doStartTag()

当 Container 编译遇到自定义标签的起始标签时，如：<Mytag>，它会根据 doStartTag() 中的程序内容做数据的处理，并且 doStartTag() 必须回传一个整数值，用来说明程序流程应该如何执行。整数值的可能性有两种，分别是 Tag.SKIP_BODY 和 Tag.EVAL_BODY_INCLUDE（见表 15-3）。

表 15-3

回传值	说 明
Tag.SKIP_BODY	表示本体会内容会被忽略，将控制权转移给 doEndTag()
Tag.EVAL_BODY_INCLUDE	表示本体会内容会正常执行显示，但是不对本体会内容做任何的计算、处理

笔者举个例子来说明：

JSP2.0 技术手册

```
<prefix:Mytag>
  Hello Tag Library ← 本体内容
  .....
  .....
</prefix:Mytag>
```

假若<Mytag>的 doStartTag() 方法回传 SKIP_BODY 时, 则 Hello Tag Library 字符串将不会显示在网页上; 若回传 EVAL_BODY_INCLUDE 时, 则 Hello Tag Library 字符串将被正常执行, 能够在网页上显示。

■ doAfterBody()

doAfterBody() 方法是用来重复执行自定义标签的本体内容。它会回传两种可能的整数值: Tag.SKIP_BODY 和 IterationTag.EVAL_BODY_AGAIN (见表 15-4)。

表 15-4

回传值	说 明
Tag.SKIP_BODY	表示本体内容会被忽略, 将控制权转移给 doEndTag()
IterationTag.EVAL_BODY_AGAIN	表示本体内容将重复执行, 并且再度调用 doAfterBody(), 如此循环下去, 直到回传 SKIP_BODY 为止

■ doEndTag()

doEndTag() 方法是处理自定义标签的结束标签, 如: </Mytag>, 它会根据 doEndTag() 中的程序内容做处理, 并且它也必须回传一个整数值。doEndTag() 一样也有两种可能的整数值: Tag.SKIP_PAGE 和 Tag.EVAL_PAGE (见表 15-5)。

表 15-5

回传值	说 明
Tag.SKIP_PAGE	表示 JSP 网页的执行应该马上停止, 所有在网页上的内容, 包括 JSP 的程序和静态文件都应该马上被忽略, 任何输出应该马上回传到用户的浏览器上
Tag.EVAL_PAGE	表示 JSP 网页能够正常执行

不管回传值是 Tag.SKIP_PAGE 或 Tag.EVAL_PAGE, doEndTag() 最后都会调用 release() 方法。

■ release()

release() 将标签处理类所产生或是获得的资源都释放, 并且重新设定标签处理类的初始状态, 然后这个标签处理类再放至资源池(Resource Pool)中, 等待下一次的使用。

最后笔者再补充一个重点, 假设你的自定义标签有属性值时, 如下:

```
<prefix:Mytag attribute1= "value1">
```

```
</prefix:Mytag>
```

这里提供一项和 JavaBean 一样取得(getter)和设定(setter)的机制, 以上述范例来说, 我们只要在标签处理类中加入 `setAttribute1()` 和 `getAttribute1()` 两个方法, 就能够取得和设定 `attribute1` 的值。下面是一段范例程序, 笔者假设 `attribute1` 的类型是 `int`。

```
private int attribute1;

public void setAttribute1(int value1) {
    this.attribute1 = value1;
}

public int getAttribute1() {
    return this.attribute1;
}
```

15-3-3 BodyTagSupport 类

`BodyTagSupport` 类是继承于 `TagSupport` 类, 并且实现 `BodyTag` 接口。`BodyTagSupport` 类和 `TagSupport` 类最大的区别在于: `BodyTagSupport` 类多实现 `BodyTag` 接口, 因此 `BodyTagSupport` 类可以处理本体内容的数据。图 15-5 为实现 `BodyTag` 接口标签的流程图:

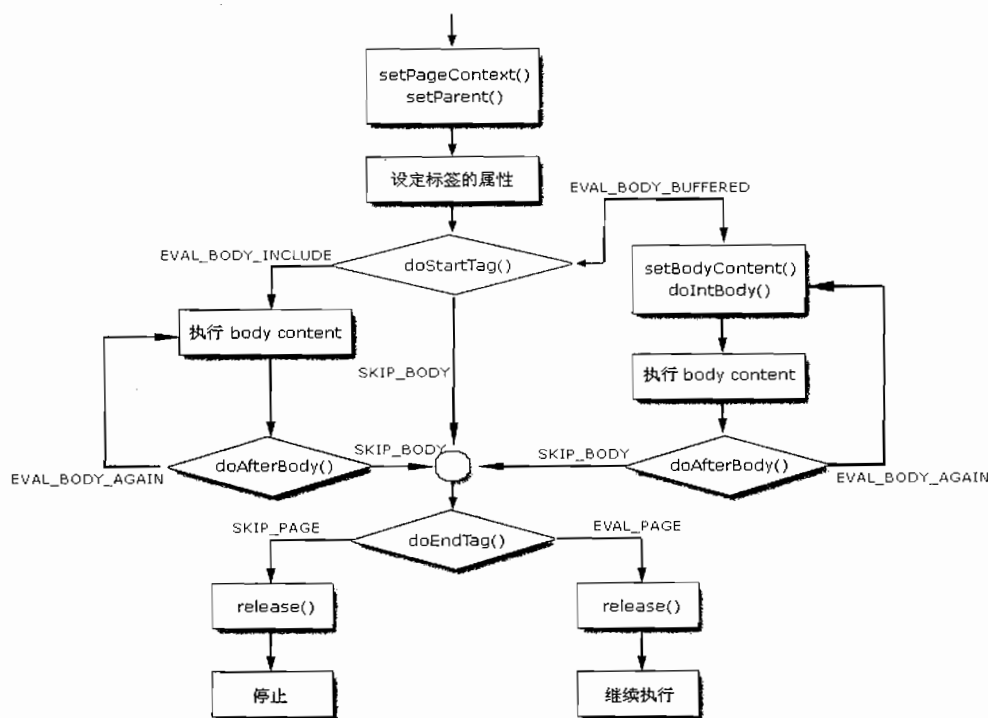


图 15-5 实现 `BodyTag` 接口标签的流程图

BodyTagSupport 类除了有 doStartTag()、doAfterBody()和 doEndTag()三个方法之外,还增加了 getBodyContent()、setBodyContent()和 doInitBody() 等方法。

```
public int doStartTag( ) throws JspException
public int doEndTag( ) throws JspException
public int doAfterBody( ) throws JspException
public BodyContent getBodyContent( )
public void setBodyContent(BodyContent b)
public void doInitBody( ) throws JspException
```

doStartTag()方法的功用和 TagSupport 类一样,不过它回传的整数值和 TagSupport 类不太一样,由图 15-5 可知,doStartTag()回传的整数值可为 Tag.SKIP_BODY、BodyTag.EVAL_BODY_BUFFERED 和 Tag.EVAL_BODY_INCLUDE (见表 15-6)。

表 15-6

回传值	说 明
Tag.SKIP_BODY	表示本体会内容会被忽略,将控制权转移给 doEndTag()
BodyTag.EVAL_BODY_BUFFERED	表示标签的本体内容应该被处理,并且处理的结果必须储存在 BodyContent 类中
Tag.EVAL_BODY_INCLUDE	表示本体会内容会正常执行显示,但是不对本体内容做任何的运算、处理

为了处理标签之间的本体内容,首先要产生(若之前执行过时,直接从资源池中取得)BodyContent 类的实体,如同调用 setBodyContent(BodyContent b)一样,然后 Container 再调用 doInitBody()方法,让用户能够新增初始值。

处理完标签的本体内容后,Container 会调用 doAfterBody()方法,重复处理标签的本体内容。doAfterBody()方法也必须回传整数值,其中可能性有两种,分别为: Tag.SKIP_BODY 和 BodyTag.EVAL_BODY_AGAIN。举个例子说明:

```
public class doAfterBodySample extends BodyTagSupport
{
    public int doAfterTag( )
    {
        BodyContent body = getBodyContent( );
        String content = body.getString( );
        JspWriter out = body.getEnclosingWriter( );

        int count = 3
        out.println(content);
        if (count>0)
        {
            count = count - 1;
            return(EVAL_BODY_AGAIN);
        }
        else
        {
            return(SKIP_BODY);
        }
    }
}
```

JSP2.0 技术手册

```

    }
}

<prefix:doAfterBodySample>
Hello , Test doAfterBody.
</ prefix:doAfterBodySample >

```

执行结果会重复显示三次 Hello , Test doAfterBody.字符串。如下所示:

```

Hello , Test doAfterBody.
Hello , Test doAfterBody.
Hello , Test doAfterBody.

```

最后, doEndTag()方法就和之前的一模一样。

15-3-4 TagExtraInfo 和 VariableInfo 类

前面介绍了 TagSupport、BodyTagSupport 和 BodyContent 类, 现在为读者介绍 TagExtraInfo 和 VariableInfo 类。

笔者先用一个范例程序来说明如何使用 TagExtraInfo 和 VariableInfo 类, 顺便也了解它的功能所在。我们可以在继承 TagExtraInfo 的类中声明一些变量, 然后在 JSP 网页中取得这些变量。现在笔者对之前的 Hello 范例程序做小幅度的修改, 在 Hello 的标签处理类中加上一小段程序代码:

```
pageContext.setAttribute("Message", "Use TEI");
```

然后, 希望在 JSP 网页中直接取得属性 Message 的值:

```

<mytag:Hello />
<%= Message %>

```

完整的程序代码如下:

■ Hello_TEI.java

```

package tw.com.javaworld.CH15;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;

public class Hello_TEI extends TagSupport {

    public int doStartTag( ) {
        try {
            JspWriter out = pageContext.getOut();
            out.println("Hello World Use Tag Library");
            pageContext.setAttribute("Message", "Use TEI");
        } catch (Exception e) {
            System.out.println("Hello Tag Error : " + e);
        }
        return (SKIP_BODY);
    }
}

```

JSP2.0 技术手册

```
}
```

然后再新增一个叫 TEI 的类，它是延至 TagExtraInfo 类并且使用 getVariableInfo() 方法的，因为通过它们可以直接在 JSP 网页中取得变量值。

■ TEI.java

```
package tw.com.javaworld.CH15;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class TEI extends TagExtraInfo {

    public TEI( ) {

    }

    public VariableInfo[] getVariableInfo(TagData data) {

        VariableInfo info = new VariableInfo("Message", "String", true,
                                              VariableInfo.AT_END);
        VariableInfo[] Var = { info };
        return Var;
    }
}
```

TagExtraInfo 类提供四个方法来处理我们所声明的变量。它们分别是：getVariableInfo()、isValid()、setTagInfo() 和 getTagInfo()。

```
public VariableInfo[] getVariableInfo(TagData data)
public boolean isValid(TagData data)
public final void setTagInfo(TagInfo tagInfo)
public final TagInfo getTagInfo( )
```

TEI.java 中，我们使用 getVariableInfo() 方法回传一个 VariableInfo 类型的对象。VariableInfo 对象的构造函数有四个参数（见表 15-7）。

```
public VariableInfo(String ID, String ClassName, boolean Declare,
int Scope)
```

表 15-7

参 数	说 明
ID	变量名称
ClassName	正确的类名称
Declare	是否之前声明过此变量
Scope	变量的可用范围，可分为三种：VariableInfo.AT_BEGIN、VariableInfo.AT_END 和 VariableInfo.NESTED。VariableInfo.AT_BEGIN：表示变量的范围从起始标签<mytag>开始到 JSP 网页结束 VariableInfo.AT_END：表示变量的范围从结束标签</mytag>开始到 JSP 网页结束 VariableInfo.NESTED：表示变量的范围从起始标签<mytag>开始到结束标签</mytag>结束

■ MyTaglib.tld

```
<?xml version="1.0" encoding="UTF-8" ?>

<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/
                        j2ee web-jsptaglibrary_2_0.xsd"
  version="2.0">

  <description>My Taglib by JavaWorld.com.tw</description>
  <tlib-version>1.0</tlib-version>
  <jsp-version>2.0</jsp-version>
  <short-name>Mytaglib</short-name>
  <uri></uri>

  <tag>
    <description>Simplest example:Hello</description>
    <name>Hello</name>
    <tag-class>tw.com.javaworld.CH15.Hello</tag-class>
    <body-content>empty</body-content>
  </tag>

  <tag>
    <description>Hello TEI</description>
    <name>Hello_TEI</name>
    <tag-class>tw.com.javaworld.CH15.Hello_TEI</tag-class>
    <tei-class>tw.com.javaworld.CH15.TEI</tei-class>
    <body-content>empty</body-content>
  </tag>

</taglib>
```

MyTaglib.tld 中，笔者多新增一个标签的设定，名称为 Hello_TEI：

```
<tag>
  <description>Hello TEI</description>
  <name>Hello_TEI</name>
  <tag-class>tw.com.javaworld.CH15.Hello_TEI</tag-class>
  <tei-class>tw.com.javaworld.CH15.TEI</tei-class>
  <body-content>empty</body-content>
</tag>
```

将 Hello_TEI 标签和 Hello 标签的设定做一个比较，可以发现到 Hello_TEI 标签的设定多一个 <tei-class>：

```
<tei-class>tw.com.javaworld.CH15.TEI</tei-class>
```

在 JSP 网页部分，因为我把变量的范围设成 VariableInfo.AT_END，因此必须在结束标签之后才能取得变量。

■ Hello_TEI.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>
```

```

<%@ taglib uri="/WEB-INF/tlds/MyTaglib.tld" prefix="mytag" %>

<html>
<head>
  <title>CH15 - Hello_TEI.jsp</title>
</head>
<body>

<h1><mytag:Hello_TEI /></h1>
${ Message. }

</body>
</html>

```

Hello_TEI.jsp 的执行结果如图 15-6 所示:



图 15-6 Hello.TEI.jsp 的执行结果

笔者最后做一个总结: 假若我们想在 JSP 网页中取得标签中声明的变量时, 我们必须先做一个继承 `TagExtraInfo` 的类, 经由这个类(例如: `TEI.java`)让标签和 JSP 网页之间的变量能够互相传递。

不过这样的方式太过于麻烦, 每当我新增一个变量时, 就必须重新改写继承 `TagExtraInfo` 的类。因此在 JSP 1.2 时, 提供一个更便利的方法: 在 TLD 文件中设定标签的变量, 这样一来, 就不需要再额外使用 `TagExtraInfo` 类。

JSP 1.2 时, 新增一个 TLD 文件的元素 `<variable>`, 我们可以直接使用它来设定标签的变量。笔者将改采使用 TLD 文件的设定方式, 重新改写前面 *Hello_TEI* 范例的 *MyTaglib*:

■ *MyTaglib.tld*

```

<?xml version="1.0" encoding="UTF-8" ?>

<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/
  j2ee web-jsptaglibrary_2_0.xsd"
  version="2.0">

```

```
.....
<tag>
  <description>Hello TEI</description>
  <name>Hello_TEI</name>
  <tag-class>tw.com.javaworld.CH15.Hello_TEI</tag-class>
  <body-content>empty</body-content>

  <variable>
    <name-given>Message</name-given>
    <variable-class>String</variable-class>
    <declare>true</declare>
    <scope>AT_END</scope>
  </variable>
</tag>
.....
</taglib>
```

和之前 TLD 文件不同的地方：`<tei-class>`元素不见了，`<variable>`元素取而代之。`<variable>`元素又有四个子元素：`<name-given>`、`<variable-class>`、`<declare>`和`<scope>`。这四个元素所代表的意思和之前介绍 `VariableInfo` 对象构造函数的四个参数一样（见表 15-8）。

表 15-8

子元素	说 明
<code><name-given></code>	变量名称
<code><variable-class></code>	变量的类型，默认为 <code>java.lang.String</code>
<code><declare></code>	<code>true</code> 或 <code>false</code> 。表示变量是否要声明，默认为 <code>true</code>
<code><scope></code>	<code>AT_BEGIN</code> 、 <code>NESTED</code> 和 <code>AT_END</code> 。表示变量的范围，默认为 <code>NESTED</code>

除了变量的名称可以由我们自定义(`<name-given>`)之外，还可以由标签属性的值来决定变量的名称，使用的方法很简单，即把`<name-given>`改为`<name-from-attribute>`，例如：

```
.....
<tag>
  <name>Hello_TEI</name>
  .....
  <variable>
    <name-from-attribute>id</name-from-attribute>
    <variable-class>String</variable-class>
    <declare>true</declare>
    <scope>AT_BEGIN</scope>
  </variable>
</tag>
.....
```

这样表示变量名称为标签 `id` 属性的值，例如：

```
<h1>
  <mytag:Hello_TEI id="myMessage">
</mytag: Hello_TEI >
</h1>
```


此时变量名称为 `myMessage`。因为在 TLD 中，设定变量名称是由标签 `id` 属性的值为定。这样一来，变量名称就能动态命名，弹性也变得更好。

15-3-5 其他类

本小节介绍三个类，它们分别为 `TagLibraryInfo`、`TagInfo` 和 `TagAttributeInfo` 类，因为这些类主要是让 `Container` 使用，通常用它们来表示 TLD 文件，不过程序员很少会用到它们，除非您自己也设计一个 JSP 的 `Container`。`TagLibraryInfo` 类提供几个方法来取得和标签函数库有关的设定值，如下所示：

```
public java.lang.String getURI( )
public java.lang.String getPrefixString( )
public java.lang.String getShortName( )
public java.lang.String getReliableURN( )
public java.lang.String getInfoString( )
public java.lang.String getRequiredVersion( )
public TagInfo[] getTags( )
public TagInfo getTag(java.lang.String shortname)
```

其中 `getURI()` 会回传在 JSP 网页 `<%@ taglib %>` 中设定的 `uri` 值；`getPrefixString()` 会回传在 JSP 网页 `<%@ taglib %>` 中设定的 `prefix` 值。而 `getShortName()`、`getReliableURN()`、`getInfoString()` 和 `getRequiredVersion()` 都和 TLD 文件有关，用在取得 TLD 的设定值。`getTag()` 是取得 `TagInfo` 对象的数组。

15-4 Tag Library 范例程序

本节笔者将写几个范例程序来实现前面所介绍的内容，并且希望读者看完这几个范例程序后，更能够了解标签函数库的用处。

15-4-1 有属性的标签—`<myfont>`

首先，第一个范例是自定义一个标签，名为 `<myfont>`，它有六个属性可以供用户自行设定，它们分别为：`bgColor`、`color`、`border`、`bordercolor`、`align`、`fontSize` 和 `width`，如下：

```
<prefix:myfont  fontSize="5"  color="red">
.....
.....
</prefix:myfont>
```

这个例子主要用来教导读者写一个有属性的标签，让读者将重心放在如何在自定义标签中处理属性值。接下来看 `myfont` 标签的标签处理类 `Myfont.java`。

■ `Myfont.java`

```
package tw.com.javaworld.CH15;
```

```
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class Myfont extends TagSupport {

    private String bgColor = "#FFFFFF"; // 默认值: 白色
    private String color = "#000000"; // 字体默认黑色
    private String align = "CENTER"; // 默认居中
    private String fontSize = "3"; // 字体大小默认 3
    private String border = "0"; // 表格边宽默认为 0
    private String width = null; // 表格宽度为 null
    private String bordercolor = "#000000"; // 表格边框颜色, 默认黑色

    public void setBgColor(String newBgColor) {
        bgColor = newBgColor;
    }
    public void setColor(String newColor) {
        color = newColor;
    }
    public void setAlign(String newAlign) {
        align = newAlign;
    }
    public void setFontSize(String newFontSize) {
        fontSize = newFontSize;
    }
    public void setBorder(String newBorder) {
        border = newBorder;
    }
    public void setWidth(String newWidth) {
        width = newWidth;
    }
    public void setBordercolor(String newBordercolor) {
        bordercolor = newBordercolor;
    }

    public int doStartTag() {
        try {
            JspWriter out = pageContext.getOut();
            out.print("<table border=" + border + " bordercolor=" +
                bordercolor);
            if (width != null) {
                out.print(" WIDTH=\"" + width + "\" >");
            }
            out.print("><TD bgcolor=" + bgColor + ">");
            out.print("<div align=" + align + "><font size=" +
                fontSize + " color=" + color + "> ");
        } catch (Exception e) {
            System.out.println("Error in doStartTag of Myfont Handler
                Class: " + e);
        }
        return (EVAL_BODY_INCLUDE);
    }
}
```

JSP2.0 技术手册

```

    public int doEndTag() {
        try {
            JspWriter out = pageContext.getOut();
            out.print("</td></tr></table>");
        } catch (Exception e) {
            System.out.println("Error in doEndTag of Myfont Handler Class:"
                               + e);
        }
        return (EVAL_PAGE);
    }
}

```

从 *Myfont.java* 中可以发现, 当我们自定义的标签有属性时, 在标签处理类中可以利用和 JavaBean 一样的机制来取得属性的值。我们一开始规划<Myfont>中有七个属性: bgColor、color、border、bordercolor、align、fontSize 和 width。笔者首先定义这七个属性的默认值, 如果用户在 JSP 网页使用<Myfont>标签时没有设定这七个属性的值, 它就会自动使用默认值来做处理 (见表 15-9)。

表 15-9

bgColor	表格中的背景颜色
color	字体的颜色
border	表格边框的宽度
bordercolor	表格边框的颜色
align	字体要居中还是靠左、靠右
fontSize	字体的大小
width	表格的宽度

因为<myfont>标签并不须要处理本体内容, 因此 *Myfont.java* 继承 TagSupport 类, 然后改写 doStartTag()和 doEndTag()两个方法。

doStartTag()方法只是将属性值和一些字符串加在一起, 再用 JspWriter 类型的 out 对象显示出来, 这部分和 Hello 范例程序都差不多。最后它回传 EVAL_BODY_INCLUDE, 表示标签的本体内容能够正常执行。

```

public int doStartTag() {
    try {
        JspWriter out = pageContext.getOut();
        out.print("<table border=" + border + " bordercolor="
                  + bordercolor);
        if (width != null) {
            out.print(" WIDTH=\"" + width + "\" >");
        }
        out.print("><TD bgcolor=" + bgColor + ">");
        out.print("<div align=" + align + "><font size=" + fontSize +
                  " color=" + color + "> ");
    } catch (Exception e) {
        System.out.println("Error in doStartTag of Myfont Handler Class:"
                           + e);
    }
}

```

JSP2.0 技术手册

```

        + e);
    }

    return (EVAL_BODY_INCLUDE);
}

```

doEndTag()方法和 doStartTag()做的事情差不多, 不过这里惟一要注意的也是它的回传值 EVAL_PAGE。当它回传 EVAL_PAGE 时, 则表示 JSP 网页仍然可以继续正常执行。如果回传 SKIP_PAGE 时, 则表示 JSP 网页不再继续执行。

```

public int doEndTag() {
    try {
        JspWriter out = pageContext.getOut();
        out.print("</td></tr></table>");
    } catch (Exception e) {
        System.out.println("Error in doEndTag of Myfont Handler Class: "
            + e);
    }
    return (EVAL_PAGE);
}

```

接下来看 *MyTaglib.tld* 的设定方式:

■ *MyTaglib.tld*

```

<?xml version="1.0" encoding="UTF-8" ?>

<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/
        j2ee web-jsptaglibrary_2_0.xsd"
    version="2.0">

    <description>My Taglib by JavaWorld.com.tw</description>
    <tlib-version>1.0</tlib-version>
    <jsp-version>2.0</jsp-version>
    <short-name>Mytaglib</short-name>
    <uri></uri>

    .....

    <tag>
        <description>Example:myfont</description>
        <name>myfont</name>
        <tag-class>tw.com.javaworld.CH15.Myfont</tag-class>
        <body-content>JSP</body-content>

        <attribute>
            <name>bgColor</name>
            <required>false</required>
        </attribute>
        <attribute>
            <name>color</name>
            <required>true</required>

```

JSP2.0 技术手册

```

    </attribute>
    <attribute>
      <name>bordercolor</name>
      <required>>false</required>
    </attribute>
    <attribute>
      <name>align</name>
      <required>>false</required>
    </attribute>
    <attribute>
      <name>fontSize</name>
      <required>>false</required>
    </attribute>
    <attribute>
      <name>border</name>
      <required>>false</required>
    </attribute>
    <attribute>
      <name>width</name>
      <required>>false</required>
    </attribute>
  </tag>
</taglib>

```

上面的 *MyTaglib.tld*，笔者只列出本范例有关的 TLD 设定，接下来详细说明它们：

```

<tag>
  <description>Example:myfont</description>
  <name>myfont</name>
  <tag-class>tw.com.javaworld.CH15.Myfont</tag-class>
  <body-content>JSP</body-content>
  .....
</tag>

```

我们新增一个标签，名称为 *myfont*，它的标签处理类是 *tw.com.javaworld.CH15.Myfont*。这里有一个要特别注意的地方：我们在 *<body-content>* 元素中设为 *JSP*，表示如果本体内内容中有 *JSP* 程序时，它会被编译并且执行。

```

<tag>
  .....
  <attribute>
    <name>bgColor</name>
    <required>>false</required>
  </attribute>
  <attribute>
    <name>color</name>
    <required>>true</required>
  </attribute>
  ..... 略
</tag>

```

这是在设定<myfont>标签的属性,笔者在此只举两个 attribute 元素来说明。我们新增一个属性名称为 bgColor, required 子元素设为 false,表示用户在使用<myfont>标签时不一定要设定 bgColor 这个属性。不过 color 属性的设定,required 元素设为 true,那就表示使用<myfont>标签时一定要指定 color 的值,如下所示:

```
<prefix:myfont color="black">
.....
</prefix:myfont>
```

假若使用<myfont>标签时没指定 color 属性,以 Tomcat 5.0.16 为例,则会产生以下的错误信息(见图 15-7):

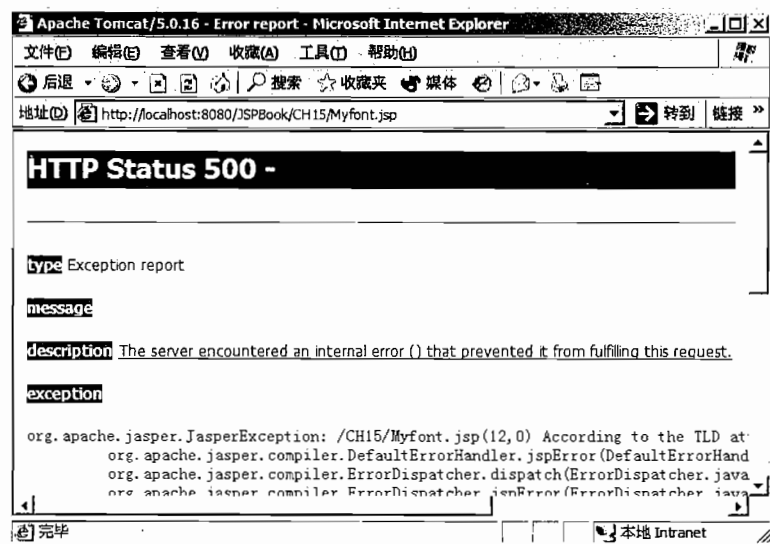


图 15-7 使用<myfont>标签时没指定 color 属性

在开发标签时,这些都要事先定义好,将来在应用时也能够得心应手。接下来看 *Myfont.jsp* 的程序。

■ *Myfont.jsp*

```
<%@ page contentType="text/html; charset=GB2312" %>
<%@ taglib uri="/WEB-INF/tlds/MyTaglib.tld" prefix="mytag" %>

<html>
<head>
  <title>CH15 - Myfont.jsp</title>
</head>
<body>

  <h2>myfont 标签</h2>

  <mytag:myfont color="Blue">
```

JSP2.0 技术手册


```

    myfont 的范例程序
</mytag:myfont><br>

<mytag:myfont color="Black" border="3" bgColor="white" >
    myfont 的范例程序
</mytag:myfont><br>

<mytag:myfont color="Blue" fontSize="6" bgColor="#C0C0C0">
    myfont 的范例程序
</mytag:myfont><br>

<mytag:myfont color="#C0C0C0" border="7" bgColor="#123456" fontSize="5">
    myfont 的范例程序
</mytag:myfont>

</body>
</html>

```

Myfont.jsp 和 *Hello.jsp* 差不多，在此不多加赘述。图 15-8 就是 *Myfont.jsp* 所执行的结果：



图 15-8 *Myfont.jsp* 的执行结果

Myfont.jsp 经过编译后所产生的 html 码如下所示：

■ *Myfont.jsp* 经编译后的 Html 码

```

<html>
<head>
  <title>CH15 - Myfont.jsp</title>
</head>
<body>

<h2>myfont 标签</h2>

```

```

<table border=0 bordercolor=#000000>
<tr><td bgcolor=#FFFFFF><div align=CENTER><font size=3 color=Blue>
myfont的范例程序
</div></td></tr></table><br>

<table border=1 bordercolor=#000000>
<tr><td bgcolor=white><div align=CENTER><font size=3 color=Black>
myfont的范例程序
</div></td></tr></table><br>

<table border=0 bordercolor=#000000>
<tr><td bgcolor=#C0C0C0><div align=CENTER><font size=6 color=Blue>
myfont的范例程序
</div></td></tr></table><br>

<table border=7 bordercolor=#000000>
<tr><td bgcolor=#123456><div align=CENTER><font size=5 color=#C0C0C0>
myfont的范例程序
</div></td></tr></table>

</body>
</html>

```

笔者附上这段 Html 码, 就是希望读者能够将它和 *Myfont.jsp* 的程序代码做一个对照, 这样就能看出 `<myfont>` 标签到底做了些事情。

15-4-2 可显示 HTML 源文件的标签——`<filter>`

之前范例是继承 `TagSupport` 类, 现在举一个继承 `BodyTagSupport` 类的范例程序。

一般来说, `<`、`>`、`"`和 `&` 的特殊符号在网页上显示时, 不是显示乱码, 就是被当做 Html 的标签, 因此我们通常需要将它们分别转成 `<`、`>`、`"`、`&`。`<filter>` 标签的功能就是将本体内容经过一个转换特殊符号的函数, 然后再显示出来。

自定义 `<filter>` 标签分为三个部分: *Filter.java*、*MyTaglib.tld* 和 *Filter.jsp*。

■ *Filter.java*

```

package tw.com.javaworld.CH15;

import javax.servlet.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class Filter extends BodyTagSupport {

    public Filter() {
    }

    public int doAfterBody() {

```

```

// 取得 body content 对象
BodyContent bc = getBodyContent();

// 取得 request 对象
ServletRequest request = pageContext.getRequest();

// 若得到 Filter 参数值为 Filter 时, 将 body content 的内容传入至 filter()
// 否则, 直接显示原始数据内容

String filter;

if (request.getParameter("Filter") != null
    && request.getParameter("Filter").equals("Filter")) {
    filter = toFilter(bc.getString());
} else {
    filter = bc.getString();
}

try {
    JspWriter out = bc.getEnclosingWriter();
    out.print(filter);
} catch (Exception e) {
    System.out.println("Error in FilterTag: " + e);
}

return (SKIP_BODY);
}

// 主要将字符串的内容中有以下特殊符号的: <、>、"、&
// 分别替换为 &lt;、&gt;、&quot;、&amp;
public String toFilter(String input) {
    StringBuffer filtered = new StringBuffer(input.length());
    char c;
    for (int i = 0; i < input.length(); i++) {
        c = input.charAt(i);
        if (c == '<') {
            filtered.append("&lt;");
        } else if (c == '>') {
            filtered.append("&gt;");
        } else if (c == '"') {
            filtered.append("&quot;");
        } else if (c == '&') {
            filtered.append("&amp;");
        } else {
            filtered.append(c);
        }
    }
    return (filtered.toString());
}
}

```

Filter.java 当中包含两个方法: `doAfterBody()` 和 `toFilter()`。`doAfterBody()` 主要是取得本

体内容并做处理，而 `toFilter()` 负责将特殊符号替换为合适的符号。

```
public int doAfterBody() {
    // 取得 body content 对象
    BodyContent bc = getBodyContent();

    // 取得 request 对象
    ServletRequest request = pageContext.getRequest();

    // 若得到 Filter 参数值为 Filter 时，将 body content 的内容传入至 toFilter()
    // 否则，直接显示原始数据内容

    String filter;

    if (request.getParameter("Filter") != null
        && request.getParameter("Filter").equals("Filter")) {
        filter = toFilter(bc.getString());
    } else {
        filter = bc.getString();
    }
    try {
        JspWriter out = bc.getEnclosingWriter();
        out.print(filter);
    } catch (Exception e) {
        System.out.println("Error in FilterTag: " + e);
    }
    return (SKIP_BODY);
}
```

`doAfterBody()` 主要做几件事情，第一，利用 `getBodyContent()` 取得 `<filter>` 标签的本体内容；第二，利用 `pageContext.getRequest()` 取得 `request` 对象，然后判断是否取得 `Filter` 的参数，若接收到 `Filter` 的参数值为“`Filter`”时，就将本体内容传入 `toFilter()` 中，执行字符替换的工作，然后显示出来；假若没有，就直接将本体内容显示出来，不借助 `toFilter()` 处理。

最后 `doAfterBody()` 回传 `SKIP_BODY`，表示本体内容只处理一次，假若回传 `EVAL_BODY_AGAIN` 时，表示本体内容将重复被处理。

```
public String toFilter(String input) {
    StringBuffer filtered = new StringBuffer(input.length());
    char c;
    for (int i = 0; i < input.length(); i++) {
        c = input.charAt(i);
        if (c == '<') {
            filtered.append("&lt;");
        } else if (c == '>') {
            filtered.append("&gt;");
        } else if (c == '"') {
            filtered.append("&quot;");
        } else if (c == '&') {
            filtered.append("&amp;");
        }
    }
    return filtered.toString();
}
```

JSP2.0 技术手册

```

    } else {
        filtered.append(c);
    }
}
return (filtered.toString());
}

```

toFilter()主要用来做字符替换工作，将 <、>、"、& 分别替代为 <、>、"、&，最后将替换后的字符串回传。其中要小心的地方是这四个替代字符的字尾都必须要有分号；做结尾。

■ MyTaglib.tld

```

<?xml version="1.0" encoding="UTF-8" ?>

<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/
    j2ee web-jsptaglibrary_2_0.xsd"
  version="2.0">

  <description>My Taglib by JavaWorld.com.tw</description>
  <tlib-version>1.0</tlib-version>
  <jsp-version>2.0</jsp-version>
  <short-name>Mytaglib</short-name>
  <uri></uri>

  .....

  <tag>
    <description>Example:filter</description>
    <name>filter</name>
    <tag-class>tw.com.javaworld.CH15.Filter</tag-class>
    <body-content>JSP</body-content>
  </tag>

</taglib>

```

MyTaglib.tld 的设定在前面也介绍过，因此不加以叙述。下面来看 Filter.jsp 的程序代码和执行结果的部分。

■ Filter.jsp

```

<%@ page contentType="text/html; charset=GB2312" %>
<%@ taglib uri="/WEB-INF/tlds/MyTaglib.tld" prefix="mytag" %>

<html>
<head>
  <title>CH15 - Filter.jsp</title>
</head>
<body>

<h2>filter 标签</h2>

```

请按下列链接, 将会把下面的 HTML 源代码显示出来

Filter


```
<mytag:filter>
<strong>范例程序三</strong><br>
<em>范例程序三</em><br>
<sub>范例程序三</sub><br>
<pre>范例程序三</pre><br>
<font color=red>
  <h3>范例程序三</h3>
</font>
</mytag:filter>

</body>
</html>
```

Filter.jsp 的执行结果如图 15-9 所示。

图 15-10 为按下 Filter 链接后的执行结果。



图 15-9 Filter.jsp 执行结果

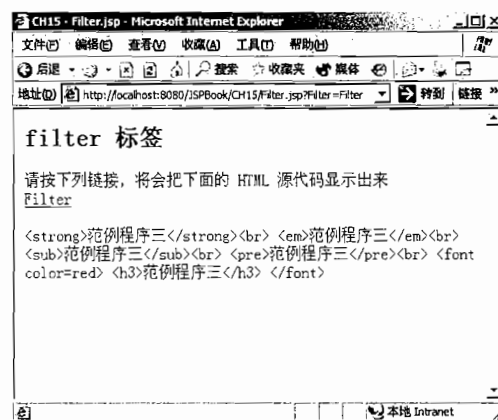


图 15-10 按下【Filter】后的结果

16

第十六章

Simple Tag 与 Tag File

第十五章已介绍了 JSP 1.2 的标签处理类：TagSupport、BodyTagSupport。本章将介绍 JSP 2.0 新增的 SimpleTagSupport 标签处理类和 DynamicAttributes 接口。实现 DynamicAttributes 接口的标签处理类，简单制作多个属性的标签。

除此之外，还将介绍 JSP 2.0 新增的一个好用的功能——Tag File。它功能强大、制作简单，因此通常用它来制作 Web 组件：模板(Template)。下面分 4 节来介绍。

- 16-1 Simple Tag
- 16-2 DynamicAttributes 接口
- 16-3 Tag File
- 16-4 Tag Library Descriptor(TLD)

JSP2.0 技术手册

16-1 Simple Tag

以往开发标签时，不是继承 `TagSupport`，就是继承 `BodyTagSupport` 类，然后实现 `doStartTag()`、`doEndTag()` 或 `doAfterBody()` 方法，而且还要搞清楚这些方法的回传值，如：`EVAL_BODY_INCLUDE`、`EVAL_PAGE`、`SKIP_BODY`、`SKIP_PAGE`、`EVAL_BODY_AGAIN` 等等。

JSP 2.0 为了简化开发标签的复杂性，因此增加了一个制作 Simple Tag 的 `SimpleTagSupport` 类。`SimpleTagSupport` 类是实现 `SimpleTag` 接口的，它只须要实现一个 `doTag()` 的方法即可，而不再需要一堆回传值。接下来我们实现一个 `HelloSimpleTag` 标签。

16-1-1 HelloSimpleTag 标签

首先我们介绍 `HelloSimpleTag` 标签的 Tag Handler——*HelloSimpleTag.java*。

■ *HelloSimpleTag.java*

```
package tw.com.javaworld.CH16;

import java.io.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class HelloSimpleTag extends SimpleTagSupport {

    public void doTag() throws JspException, IOException {

        JspWriter out = getJspContext().getOut();
        out.println("Hello Simple Tag");
    }
}
```

制作一个 Simple Tag，只须要继承 `SimpleTagSupport` 类，然后实现 `doTag()` 方法即可。`HelloSimpleTag` 标签只是单纯显示范例 `Hello Simple Tag` 的字符串。接下来再设定 TLD，延续上一章所使用的 *MyTaglib.tld*：

■ *MyTaglib.tld*

```
<?xml version="1.0" encoding="UTF-8" ?>

<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                            web-jsptaglibrary_2_0.xsd"
        version="2.0">
.....
<tag>
    <description>Hello Simple Tag</description>
    <name>HelloSimpleTag</name>
</tag>
```

JSP2.0 技术手册

```
<tag-class>tw.com.javaworld.CH16.HelloSimpleTag</tag-class>
<body-content>empty</body-content>
</tag>
.....
</taglib>
```

然后, 我们新增一个 *HelloSimpleTag.jsp* 使用 *HelloSimpleTag* 标签。

注意

Simple Tag 的 `<body-content>` 不可设为 JSP。

■ HelloSimpleTag.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>
<%@ taglib uri="/WEB-INF/tlds/MyTaglib.tld" prefix="mytag" %>

<html>
<head>
  <title>CH16 - HelloSimpleTag.jsp</title>
</head>
<body>

<h2>Simple Tag 标签</h2>

<h1><mytag:HelloSimpleTag /></h1>

</body>
</html>
```

最后, 执行 *HelloSimpleTag.jsp* 的结果如图 16-1 所示:



图 16-1 HelloSimpleTag.jsp 的执行结果

16-1-2 AddSimpleTag 标签

AddSimpleTag 标签用来将两个数字做加法的运算, 因此必须传进两个数字至 Tag Handler

之中,经过计算之后显示出结果。我们还是依照惯例,先说明 AddSimpleTag 标签的 Tag Handler——AddSimpleTag.java。

■ AddSimpleTag.java

```
package tw.com.javaworld.CH16;

import java.io.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class AddSimpleTag extends SimpleTagSupport {

    private int num1 = 0;
    private int num2 = 0;

    public void setNum1(int num1) {
        this.num1 = num1;
    }

    public void setNum2(int num2) {
        this.num2 = num2;
    }

    public void doTag() throws JspException, IOException {

        JspContext ctx = getJspContext();
        JspWriter out = ctx.getOut();

        int sum = num1 + num2;
        ctx.setAttribute("sum", Integer.toString(sum));

        out.println(num1 + " + " + num2 + " = " + sum);
    }
}
```

AddSimpleTag 标签有两个属性,分别为 num1 和 num2,代表执行加法运算的两个数字。我们使用类似 JavaBean 的 Setter 机制,将两个属性传至 AddSimpleTag.java 之中,如:setNum1 和 setNum2,然后将两数字相加之结果储存至 Page 范围之中:

```
ctx.setAttribute("sum", Integer.toString(sum));
```

并且将两数字的结果显示出来。

■ MyTaglib.tld

```
<?xml version="1.0" encoding="UTF-8" ?>
<taglib>
    .....
    <tag>
        <description>Add Simple Tag</description>
```

JSP2.0 技术手册

```

<name>Add</name>
<tag-class>tw.com.javaworld.CH16.AddSimpleTag</tag-class>
<body-content>empty</body-content>

<attribute>
  <name>num1</name>
  <required>true</required>
  <rtexprvalue>true</rtexprvalue>
</attribute>

<attribute>
  <name>num2</name>
  <required>true</required>
  <rtexprvalue>true</rtexprvalue>
</attribute>
</tag>
.....
</taglib>

```

然后，我们新增一个 *AddSimpleTag.jsp* 使用 *AddSimpleTag* 标签。

■ *AddSimpleTag.jsp*

```

<%@ page contentType="text/html; charset=GB2312" %>
<%@ taglib uri="/WEB-INF/tlds/MyTaglib.tld" prefix="mytag" %>

<html>
<head>
  <title>CH16 - AddSimpleTag.jsp</title>
</head>
<body>

<h2>AddSimpleTag 标签</h2>

<h1><mytag:Add num1="5" num2="9" /></h1>

最后结果: ${sum}

</body>
</html>

```

在 *AddSimpleTag.jsp* 中，我们使用 *AddSimpleTag* 标签，并且使用 *\${sum}* 取得储存至 Page 范围的 *sum* 变量：

```
<h1><mytag:Add num1="5" num2="9" /></h1>
```

最后结果: *\${sum}*

最后执行 *AddSimpleTag.jsp* 的结果如图 16-2 所示：

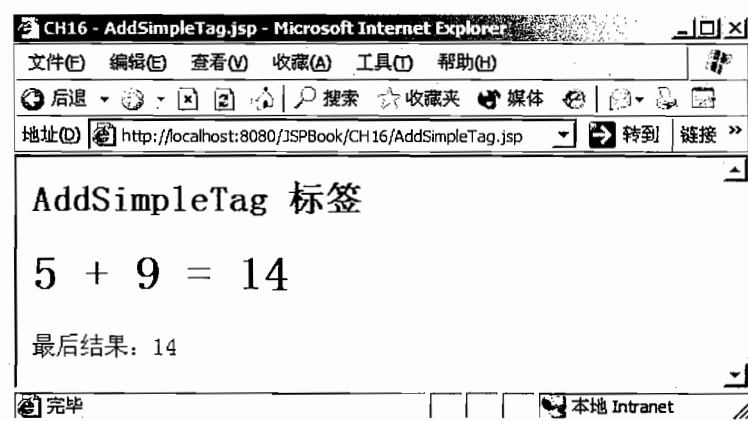


图 16-2 AddSimpleTag.jsp 的执行结果

在 Simple Tag 中, Tag Handler 传递变量给 JSP 有了更方便的方法: 即使用 JspContext 对象的 `setAttribute()` 方法。

补充

以往的做法必须使用 `TagExtraInfo` 类或者在 TLD 设定 `<variable>`, 这两种做法在“15-3-4: `TagExtraInfo` 和 `VariableInfo` 类”皆有介绍。

16-1-3 RepeatSimpleTag 标签

`RepeatSimpleTag` 标签主要用来重复显示某一段文字。首先来看 `RepeatSimpleTag` 标签的 Tag Handler——`RepeatSimpleTag.java`。

■ RepeatSimpleTag.java

```
package tw.com.javaworld.CH16;

import java.io.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class RepeatSimpleTag extends SimpleTagSupport {

    private int count = 0;
    private JspFragment fragment;

    public void setCount(int count) {
        this.count = count;
    }

    public void setFragment(JspFragment fragment) {
        this.fragment = fragment;
    }
}
```



```

public void doTag() throws JspException, IOException {

    JspContext ctx = getJspContext();
    JspWriter out = ctx.getOut();

    for(int i=0 ; i<count ; i++) {
        fragment.invoke(null);
    }
}
}

```

RepeatSimpleTag 标签有两个属性，分别为 count 和 fragment。其中 count 为 int 类型，代表重复的次数；fragment 为 JspFragment 类型，代表要重复的内容。doTag() 方法中，使用一个 for 循环重复执行 fragment.invoke(null)，invoke(null) 表示将 fragment 的内容显示出来。接下来说明 RepeatSimpleTag 标签的 TLD 设定：

■ MyTaglib.tld

```

<?xml version="1.0" encoding="UTF-8" ?>
<taglib>
.....
<tag>
    <description>Repeat Simple Tag</description>
    <name>Repeat</name>
    <tag-class>tw.com.javaworld.CH16.RepeatSimpleTag</tag-class>
    <body-content>empty</body-content>

    <attribute>
        <name>count</name>
        <required>true</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>

    <attribute>
        <name>fragment</name>
        <required>true</required>
        <fragment>true</fragment>
    </attribute>
</tag>
.....
</taglib>

```

这里惟一要注意的地方是设定 fragment 属性时，必须加上 <fragment>true</fragment>。然后，新增一个 RepeatSimpleTag.jsp 使用 RepeatSimpleTag 标签。

■ RepeatSimpleTag.jsp

```

<%@ page contentType="text/html; charset=GB2312" %>
<%@ taglib uri="/WEB-INF/tlds/MyTaglib.tld" prefix="mytag" %>

<html>
<head>
    <title>CH16 - RepeatSimpleTag.jsp</title>

```

```
</head>
<body>

<h2>RepeatSimpleTag 标签</h2>

<mytag:Repeat count="5" >
  <jsp:attribute name="fragment">
    重复执行 ....<br>
  </jsp:attribute>
</mytag:Repeat>
</body>
</html>
```

在 *RepeatSimpleTag.jsp* 中, 我们使用 *RepeatSimpleTag* 标签, 并且使用 *<jsp:attribute>* 来设定 *fragment* 属性的值:

```
<mytag:Repeat count="5" >
  <jsp:attribute name="fragment">
    重复执行 ....<br>
  </jsp:attribute>
</mytag:Repeat>
```

最后执行 *RepeatSimpleTag.jsp* 的结果如图 16-3 所示:

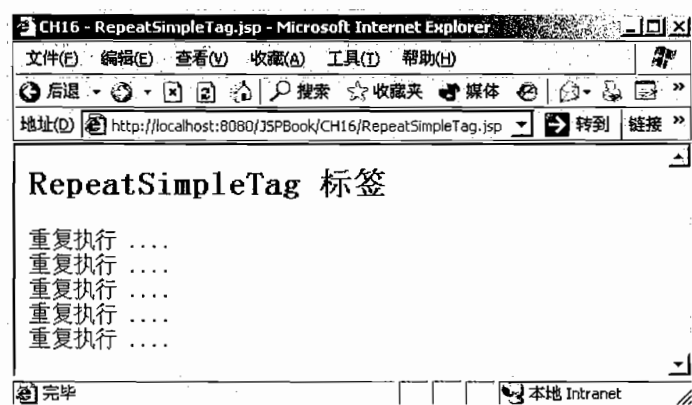


图 16-3 RepeatSimpleTag.jsp 的执行结果

16-2 DynamicAttributes 接口

JSP 2.0 新增 *DynamicAttributes* 接口。只要制作的标签实现 *DynamicAttributes* 接口, 此标签就具有动态属性的功能, 笔者直接用范例来说明。之前制作一个 *AddSimpleTag* 标签, 它只能做两个数字的加法运算, 假若我们要做多个数字的累加运算, 则 *AddSimpleTag* 标签只要实现 *DynamicAttributes* 接口即可做到。

实现 *DynamicAttributes* 接口, 必须实现 *setDynamicAttribute()* 方法, 此方法用来接收动态属性。接下来笔者将把 *AddSimpleTag* 标签制作成 *DynamicAdd* 标签, 下面来看 Tag

JSP2.0 技术手册

Handler — *DynamicAdd.java*.

■ *DynamicAdd.java*

```
package tw.com.javaworld.CH16;

import java.io.*;
import java.util.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class DynamicAdd extends SimpleTagSupport implements
    DynamicAttributes {

    private ArrayList keys = new ArrayList();
    private ArrayList values = new ArrayList();

    public void doTag() throws JspException, IOException {

        JspContext ctx = getJspContext();
        JspWriter out = ctx.getOut();

        float num = 0;
        float sum = Float.parseFloat((String)values.get(0));
        out.print(sum);

        for (int i = 1 ; i < keys.size() ; i++) {
            String temp = (String)values.get(i);
            num = Float.parseFloat(temp);
            sum = sum + num;
            out.print(" + " + num);
        }

        out.print(" = " + sum);
        ctx.setAttribute("sum", Float.toString(sum));
    }

    public void setDynamicAttribute(String uri, String name, Object value)
    throws JspException {
        keys.add(name);
        values.add(value);
    }
}
```

DynamicAdd.java 声明两个 `ArrayList`，用来接收动态属性，而 `setDynamicAttribute()` 方法就是用来接收动态属性的，其中 `name` 为属性的名称；`value` 为属性的值：

```
public void setDynamicAttribute(String uri, String name, Object value)
    throws JspException {
    keys.add(name);
    values.add(value);
}
```

接收到所有动态属性之后，然后就是将这些属性值相加，并且显示出来。接下来说明 DynamicAdd 标签的 TLD 设定：

■ MyTaglib.tld

```
<?xml version="1.0" encoding="UTF-8" ?>
<taglib>
    .....
    <tag>
        <description>DynamicAttribute</description>
        <name>DynAdd</name>
        <tag-class>tw.com.javaworld.CH16.DynamicAdd</tag-class>
        <body-content>empty</body-content>

        <dynamic-attributes>true</dynamic-attributes>
    </tag>
    .....
</taglib>
```

这里惟一要注意的地方是必须加上<dynamic-attributes>true</dynamic-attributes>。然后，新增一个 *DynamicAdd.jsp* 使用 DynamicAdd 标签。

■ DynamicAdd.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>
<%@ taglib uri="/WEB-INF/tlds/MyTaglib.tld" prefix="mytag" %>

<html>
<head>
    <title>CH16 - DynamicAdd.jsp</title>
</head>
<body>

    <h2>DynamicAdd 标签</h2>

    <h1><mytag:DynAdd num1="5.37" num2="9.11" num3="55.09" /></h1>

</body>
</html>
```

在 *DynamicAdd.jsp* 中，我们使用 DynamicAdd 标签，并且设定三个属性 num1、num2 和 num3，它们的值分别为 5.37、9.11 和 55.09：

```
<mytag:DynAdd num1="5.37" num2="9.11" num3="55.09" />
```

最后，执行 *DynamicAdd.jsp* 的结果如图 16-4 所示：

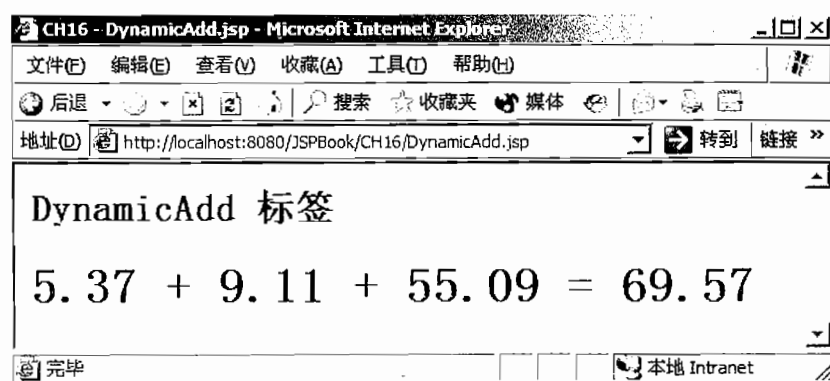


图 16-4 DynamicAdd.jsp 的执行结果

16-3 Tag File

Tag File 是 JSP 2.0 新增的主要功能之一。根据 JSP 2.0 规范中的定义：所谓的 Tag File 就是让 JSP 网页开发人员可以直接使用 JSP 语法来制作标签，而不须了解 Java 语言。

所有 Tag File 文件的扩展名都为 *.tag* 或 *.tagx*。假若 Tag File 包含其他完整或片段的 Tag File，JSP 2.0 规范建议扩展名为 *.tagf*。

接下来我们直接看简单的 Tag File 范例：*Hello.tag*、*HelloTagFile.jsp*。

■ Hello.tag

```
<%@ tag pageEncoding="GB2312" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<c:out value="Hello ~ JSP 2.0 技术手册 " />
```

Hello.tag 的内容很简单，只有显示出“Hello ~ JSP 2.0 技术手册”。*Hello.tag* 产生之后，通常会放置在 *WEB-INF/tags/* 目录下，这是 Tag File 默认的位置。接下来看 *HelloTagFile.jsp* 如何调用 *Hello.tag*。

补充

`<%@ tag pageEncoding="GB2312" %>` 用来指定 Tag File 的编码方式，若没有指定时，默认使用 ISO8859-1。

■ HelloTagFile.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>
<%@ taglib prefix="JSPBook" tagdir="/WEB-INF/tags/" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<html>
<head>
  <title>CH16 - HelloTagFile.jsp</title>
```

JSP2.0 技术手册

```
</head>
<body>

<h2>Tag File 范例</h2>

<c:out value="使用 <JSPBook:Hello /> 打印出 欢迎词" />
<h1><JSPBook:Hello /></h1>

</body>
</html>
```

设定 Hello Tag File 的 prefix 和 tagdir。其中 prefix 之前介绍过(15-2-3); tagdir 是指 Tag File 的所在位置。如下:

```
<%@ taglib prefix="JSPBook" tagdir="/WEB-INF/tags/" %>
```

调用 Hello Tag File 的方式和之前调用标签的方式类似: prefix:Tag File 名称, 如: <JSPBook:Hello />。HelloTagFile.jsp 的执行结果如图 16-5。

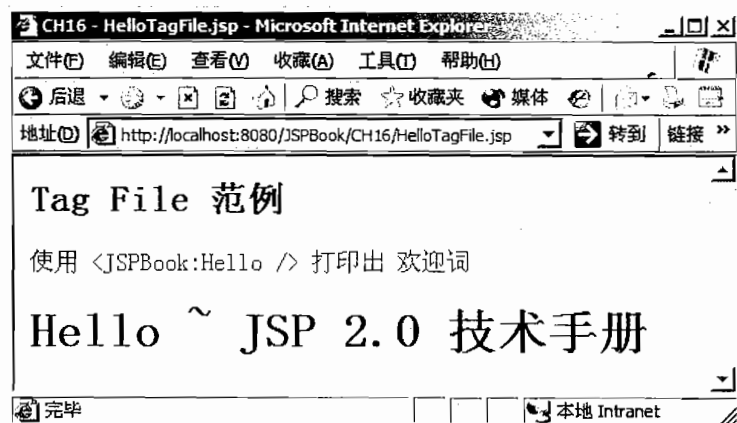


图 16-5 HelloTagFile.jsp 的执行结果

读者看完上述的范例, 是不是觉得使用 Tag File 来制作标签十分简单、方便。因为可以无须了解一堆类(如: TagSupport、BodyTagSupport 等等)、接口(如: BodyTag、SimpleTag 等等)就能制作出我们想要的功能。不过先别急着往下看, 笔者先来说明 Tag File 的底层机制。

最快了解 Tag File 底层机制的方法就是直接看 Tag File 到底被编译成什么东西。我们首先到 {Tomcat_Install}\work\ 的目录找到 JSPBook 的目录, 然后再往下找就会发现一个 tag 的目录, 其中有一个名为 Hello_tag.java 的文件, 这就是 Hello.tag 被转译的 Java 源代码, 笔者节录其中主要程序:

■ Hello_tag.java

```
package org.apache.jsp.tag.web;

import javax.servlet.*;
```

JSP2.0 技术手册


```

import javax.servlet.http.*;
import javax.servlet.jsp.*;

public final class Hello_tag extends javax.servlet.jsp.tagext.
    SimpleTagSupport
    implements org.apache.jasper.runtime.JspSourceDependent {

    private void _jspInit(ServletConfig config) {
        .....
    }

    public void _jspDestroy() {
        .....
    }

    public void doTag() throws JspException, java.io.IOException {
        .....
        _jspInit(config);

        try {
            out.write("\r\n\r\n\r\n");
            if (_jspx_meth_c_out_0(pageContext))
                return;
        } catch( Throwable t ) {
            .....
        } finally {
            .....
        }
    }

    private boolean _jspx_meth_c_out_0(PageContext pageContext) throws
        Throwable {
        .....
        _jspx_th_c_out_0.setValue(new String("Hello ~ JSP 2.0 技术手册"));
        .....
    }
}

```

由 *Hello_tag.java* 程序可以发现：原来 Tag File 就是 Simple Tag，Hello_tag 继承 SimpleTagSupport 类，证据如下：

```

public final class Hello_tag extends
    javax.servlet.jsp.tagext.SimpleTagSupport
    implements org.apache.jasper.runtime.JspSourceDependent {
    .....

    public void doTag() throws JspException, java.io.IOException {
        .....
    }
}

```

由此可知，Tag File 底层根本就是 Simple Tag，只不过 Tag File 的使用比 Simple Tag 更方便和快速。这也是为什么笔者会在介绍 Tag File 之前先介绍 Simple Tag 的原因。

JSP2.0 技术手册

16-3-1 Tag File 的语法

上面提到 Tag File 是使用 JSP 语法来制作的标签，因此 JSP 大部分的语法如 Scripting、EL，都能在 Tag File 中使用，不过两者还是有一些差别：

- (1) 有些指令(Directives)在 Tag File 不能使用，如：page，但 Tag File 多了 tag 的指令。这部分将在后续章节中介绍。
 - (2) <jsp:invoke>和<jsp:doBody>两个 actions 能在 Tag File 中使用。这部分将在后续章节中介绍。
- Tag File 一样可以使用隐含对象，不过 Tag File 只能使用以下七个隐含对象（见表 16-1）。

表 16-1

隐含对象	类 型	说 明
request	javax.servlet.http.HttpServletRequest	请求端信息
response	javax.servlet.http.HttpServletResponse	响应端信息
jspContext	javax.servlet.jsp.JspContext	表示此 Tag File 的 JspContext
session	javax.servlet.http.HttpSession	在同联机中所产生的 session 数据，目前只对 HTTP 协议有意义
application	javax.servlet.ServletContext	如同调用 getServletConfig().getServletContext()
out	javax.servlet.jsp.JspWriter	数据流的标准输出
config	javax.servlet.ServletConfig	表示此 JSP 的 ServletConfig

上述七个隐含对象比 JSP 网页的隐含对象(请参阅“第五章：隐含对象 (Implicit Object)”)少了三个，分别为 pageContext、page 和 exception，不过多了一个 jspContext 的隐含对象。在这里笔者只介绍 jspContext 隐含对象，至于其他六个隐含对象和之前介绍的一样，读者可以参阅“第五章：隐含对象 (Implicit Object)”。

JspContext 隐含对象的类型为 javax.servlet.jsp.JspContext，它主要用来管理所有范围的变量。JspContext 提供下列的方法来管理变量（见表 16-2）。

表 16-2

方 法	说 明
void setAttribute(String name, Object value)	设定 name 属性的值为 value
void setAttribute(String name, Object value, int scope)	设定 name 属性在 scope 范围的值为 value
Object getAttribute(String name)	取得 name 属性的值
Object getAttribute(String name, int scope)	取得 name 属性在 scope 范围的值
Object findAttribute(String name)	依照 page、request、session 和 application 顺序寻找 name 属性的值
void removeAttribute(String name)	移除 name 属性
void removeAttribute(String name, int scope)	移除在 scope 范围的 name 属性
int getAttributesScope(String name)	取得 name 属性的范围
Enumeration getAttributeNamesInScope(int scope)	取得所有在 scope 范围的属性

16-3-2 Tag File 的指令

Tag File 主要有五个指令，如表 16-3 所示：

表 16-3

指 令	说 明
tag	如同 JSP 网页的 page 指令
taglib	如同 JSP 网页的 taglib 指令
include	如同 JSP 网页的 include 指令
attribute	用来设定自定义标签的属性
variable	用来设定 Tag File 的变量，此变量可以回传至 JSP 网页

● tag 指令

tag 指令如同 JSP 网页的 page 指令，用来设定 Tag File。

语法

```
<%@ tag tag_directive_attr_list %>
```

```
tag_directive_attr_list ::=
```

```
{ display-name = "display-name" }
{ body-content = "scriptless | tagdependent | empty" }
{ dynamic-attributes = "name" }
{ small-icon = "small-icon" }
{ large-icon = "large-icon" }
{ description = "description" }
{ example = "example" }
{ language = "scriptingLanguage" }
{ import = "importList" }
{ pageEncoding = "peinfo" }
{ isELIgnored = "true | false" }
```

属性

名 称	说 明	必须
display-name	图形化开发工具显示<display-name>所指定的名称，例如： <description>MyTag</description>	N
body-content	它可能的值有三种，分别为：empty、scriptless 和 tagdependent。empty 表示标签中没有本体内容。scriptless 表示标签中的本体内容可以为 EL、JSP action 元素，但是不可为 JSP scripting 元素。tagdependent 表示标签中的本体内容交由 tag 自己去处理。默认值为 scriptless	N
dynamic-attributes	设定 Tag File 动态属性的名称。当 dynamic-attributes 设定时，将会产生一个 Map 类型的集合对象，用来存放属性的名称和值	N
small-icon	在图形化开发工具显示<small-icon>所指定的 TLD 相对路径之一 GIF 或 JPEG 的小图标，大小为 16 × 16	N

JSP2.0 技术手册

续表

名 称	说 明	必须
large-icon	在图形化开发工具显示<large-icon>所指定的 TLD 相对路径之一 GIF 或 JPEG 的大图标，大小为 32×32	N
description	用来说明此 Tag File 的相关信息	N
example	用来增加更多的标签使用说明，包括标签应用时的范例	N
language	与 page 指令的 language 属性相同	N
import	与 page 指令的 import 属性相同	N
pageEncoding	与 page 指令的 pageEncoding 属性相同	N
isELIgnored	与 page 指令的 isELIgnored 属性相同	N

范例

```
<%@ tag display-name="Addition" body-content="scriptless"
    dynamic-attributes="dyn"
    small-icon="/WEB-INF/sample-small.jpg"
    large-icon="/WEB-INF/sample-large.jpg"
    pageEncoding = "GB2312"
    description="Sample usage of tag directive" %>
```

● attribute 指令

attribute 指令主要用来设定自定义标签的属性。

语法

```
<%@ attribute attribute_directive_attr_list %>

attribute_directive_attr_list ::=
    name = "attribute-name"
    { required = "true / false" }
    { fragment = "true / false" }
    { rtexprvalue = "true / false" }
    { type = "type" }
    { description = "description" }
```

属性

名 称	说 明	必须
name	属性名称	Y
required	此属性是否为必要，默认值为 false	N
fragment	此属性是否为 fragment，默认值为 false	N
rtexprvalue	属性值是否可以为 run-time 表达式。假若设为 true，表示属性值可用动态的方式来指定，如：<Mytag:Handler num="\${param.num}" />，假若设为 false，则一定要用静态的方式来指定属性值	N
type	此属性的类型，默认值为 java.lang.String	N
description	用来说明此属性的相关信息	N

范例

```
<%@ attribute name = "x" required = "true" fragment = "false"
      rtexprvalue = "false" type = "java.lang.Integer"
      description = "The first operand" %>
<%@ attribute name = "y" type = "java.lang.Integer" %>
<%@ attribute name = "prompt" fragment = "true" %>
```

● variable 指令

variable 指令主要用来设定 Tag File 的变量。

语法

```
<%@ variable variable_directive_attr_list %>

variable_directive_attr_list ::=
    ( name-given = "output-name"
      | ( name-from-attribute = "attr-name"
          alias = "local-name" )
      )
    { variable-class = "output-type" }
    { declare = "true / false" }
    { scope = "AT_BEGIN / AT_END / NESTED" }
    { description = "description" }
```

属性

名 称	说 明	必 须
name-given	<name-given>表示直接指定变量的名称。15-3-4 小节有更详细的说明。	N
name-from-attribute	<name-from-attribute>表示以自定义标签的某个属性值为变量名称。15-3-4 小节有更详细的说明	N
alias	声明一局部范围属性，用来接收变量的值	N
variable-class	变量的类名称，默认值为 java.lang.String	N
declare	此变量是否声明默认值为 true	N
scope	此变量的范围。范围分为：AT_BEGIN、AT_END 和 NESTED，默认值为 NESTED	N
description	用来说明此变量的相关信息	N

范例

```
<%@ variable name-given="sum" variable-class="java.lang.Integer"
      scope="NESTED" declare="true"
      description="The sum of the two operands" %>

<%@ variable name-given="op1" variable-class="java.lang.Integer"
      description="The first operand"%>
<%@ variable name-from-attribute="var" alias="result" %>
```


这里笔者举一个显示目前时间的范例：*NowDate.tag* 和 *NowDate.jsp*。

■ *NowDate.tag*

```
<%@ tag pageEncoding="GB2312" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ tag import="java.util.Date,java.text.DateFormat" %>
<%@ variable name-given="fullDate" %>
<%@ variable name-given="longDate" %>
<%@ variable name-given="mediumDate" %>
<%@ variable name-given="shortDate" scope="AT_END" %>
<%
    Date now = new Date(System.currentTimeMillis());
    DateFormat fullFormat = DateFormat.getDateInstance(DateFormat.FULL);
    DateFormat longFormat = DateFormat.getDateInstance(DateFormat.LONG);
    DateFormat mediumFormat =
        DateFormat.getDateInstance(DateFormat.MEDIUM);
    DateFormat shortFormat = DateFormat.getDateInstance(DateFormat.SHORT);
    jspContext.setAttribute("fullDate", fullFormat.format(now));
    jspContext.setAttribute("longDate", longFormat.format(now));
    jspContext.setAttribute("mediumDate", mediumFormat.format(now));
    jspContext.setAttribute("shortDate", shortFormat.format(now));
%>
<jsp:doBody/>
```

NowDate.tag 中声明四个变量，分别为 *fullDate*、*longDate*、*mediumDate* 和 *shortDate*，其中除了 *shortDate* 为 *AT_END* 范围之外，其他皆为 *NESTED* 范围。

```
<%@ variable name-given="fullDate" %>
<%@ variable name-given="longDate" %>
<%@ variable name-given="mediumDate" %>
<%@ variable name-given="shortDate" scope="AT_END" %>
```

笔者再将四种日期格式的数据 *fullFormat.format(now)*、*longFormat.format(now)*、*mediumFormat.format(now)* 和 *shortFormat.format(now)* 利用 *jspContext.setAttribute()* 的方法分别来储存到这四个变量。

最后再调用 *<jsp:doBody />* 执行标签的内容，它和 *<jsp:invoke>* 的功能很类似，并且它们都只能使用在 *Tag File* 之中，而不可以使用在 *JSP* 网页里。在随后的章节中，会更详细地介绍这两个 actions。

■ *NowDate.jsp*

```
<%@ page contentType="text/html; charset=GB2312" %>
<%@ taglib prefix="JSPBook" tagdir="/WEB-INF/tags/" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<html>
<head>
    <title>CH16 - NowDate.jsp</title>
</head>
<body>
```



```
<h2>Tag File 范例</h2>
```

```
<JSPBook:NowDate>
```

```
    现在时间 In full format: ${fullDate}<br>
```

```
    现在时间 In long format: ${longDate}<br>
```

```
    现在时间 In medium format: ${mediumDate}<br>
```

```
    现在时间 In short format: ${shortDate}<br>
```

```
</JSPBook:NowDate>
```

```
Now Date is: ${shortDate}...
```

```
</body>
```

```
</html>
```

NowDate.jsp 直接调用执行 *NowDate* Tag File, 并且使用 EL 来取得 *NowDate* Tag File 的四个变量。因为 *fullDate*、*longDate* 和 *mediumDate* 的变量范围都是 NESTED, 所以可以在 *<JSPBook:NowDate>* 和 *</JSPBook:NowDate>* 的中间顺利取得数据, 不过 *shortDate* 的变量范围是 AT_END, 因此无法顺利取得, 必须在 *</JSPBook:NowDate>* 标签之后才能取得 *shortDate* 变量的值。图 16-6 为 *NowDate.jsp* 的执行结果:

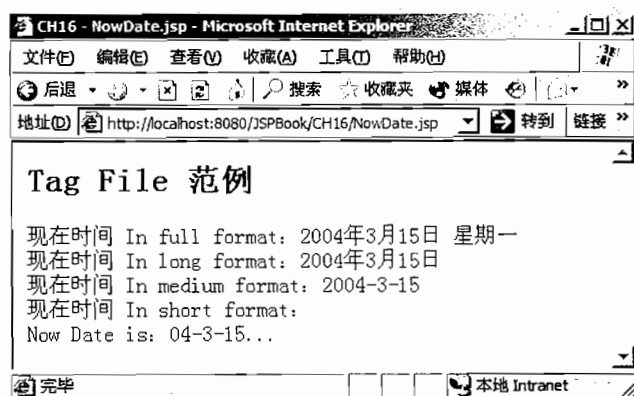


图 16-6 *NowDate.jsp* 的执行结果

假若 *NowDate.tag* 拿掉 *<jsp:doBody/>* 时, 则 *NowDate.jsp* 的执行结果则会变为图 16-7:



图 16-7 *NowDate.tag* 拿掉 *<jsp:doBody/>* 时, *NowDate.jsp* 的执行结果

最后笔者使用 Tag File 改写上一章节的加法器范例：DynAdd.jsp 和 DynAdd.tag。

■ DynAdd.tag

```
<%@ tag pageEncoding="GB2312" %>
<%@ tag dynamic-attributes="numColumn" %>
<%@ attribute name="great" fragment="true" %>
<%@ attribute name="less" fragment="true" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ variable name="sum" variable-class="java.lang.Object" %>

<c:if test="${not empty numColumn}" >
  <c:forEach items="${numColumn}" var="num">
    <c:set var="sum" value="${num.value + sum}" />
  </c:forEach>
  <c:if test="${sum >= 1000}" >
    <jsp:invoke fragment="great" />
  </c:if>
  <c:if test="${sum < 1000}" >
    <jsp:invoke fragment="less" />
  </c:if>
</c:if>
```

DynAdd.tag 声明三个属性(numColumn、great 和 less)及一个变量(sum)。其中 numColumn 为动态属性、great 和 less 皆为 fragment 属性，而 sum 变量的类型为 java.lang.Object，范围为 NESTED。

注意

倘若 sum 变量的类型没有设为 java.lang.Object，程序编译执行时会产生错误。

```
<%@ tag dynamic-attributes="numColumn" %>
<%@ attribute name="great" fragment="true" %>
<%@ attribute name="less" fragment="true" %>
<%@ variable name="sum" variable-class="java.lang.Object" %>
```

声明完毕之后，首先判断 numColumn 是否有数据，如果有数据时，因为 numColumn 的类型为 Map，所以可以直接使用<c:forEach>将 numColumn 所有的值取出来做累加的动作，最后再存入 sum 变量之中。

```
<c:if test="${not empty numColumn}" >
  <c:forEach items="${numColumn}" var="num">
    <c:set var="sum" value="${num.value + sum}" />
  </c:forEach>
  ....
</c:if>
```

最后倘若 sum 变量(即总和)大于等于 1000 时，就调用 great fragment；若小于 1000 时，则调用 less fragment。

```
<c:if test="${sum >= 1000}" >
  <jsp:invoke fragment="great" />
</c:if>
<c:if test="${sum < 1000}" >
```

```
<jsp:invoke fragment="less" />
</c:if>
```

我们再来看看 *DynAdd.jsp* 是如何使用 DynAdd Tag File 的。

■ *DynAdd.jsp*

```
<%@ page contentType="text/html; charset=GB2312" %>
<%@ taglib prefix="JSPBook" tagdir="/WEB-INF/tags/" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<html>
<head>
  <title>CH16 - DynAdd.jsp</title>
</head>
<body>

<h2>Tag File 范例</h2>

<JSPBook:DynAdd num1="111" num2="222" num3="444" >

  <jsp:attribute name="great">
    <font color="red">SUM: ${sum} ...</red>
  </jsp:attribute>
  <jsp:attribute name="less">
    <font color="blue">SUM: ${sum} ...</red>
  </jsp:attribute>
</JSPBook:DynAdd>

</body>
</html>
```

DynAdd.jsp 调用 DynAdd Tag File 时, 笔者加了四个属性: num1、num2、num3 和 num4。虽然其中 num4 的做法有些不同, 但是意思和 num1.2.3 一样, 这只不过是笔者想让读者知道也可以使用<jsp:attribute>来设定属性。

再来设定两个 fragment: <jsp:attribute name="great">和<jsp:attribute name="less">。两者只有 color 的值不一样, 一个为 red, 另一个为 blue。这样表示假若 great fragment 被 invoke 时, 会显示红色的 SUM: XXXX; 若 less fragment 被 invoke 时, 会显示蓝色的 SUM: XXXX。图 16-8 为 *DynAdd.jsp* 的执行结果, 其中 SUM: 1332 的颜色为红色。



图 16-8 *DynAdd.jsp* 的执行结果

假若我拿掉 *DynAdd.jsp* 中的 `<jsp:attribute name="num4">555</jsp:attribute>` 时, 则 SUM: 777 的颜色为蓝色。如图 16-9 所示:



图 16-9 sum 小于 1000 时, SUM: 777 的颜色为蓝色

16-4 Tag Library Descriptor (TLD)

Tag Library Descriptor 文件, 简称 TLD, 它是 XML 文件格式, 因此它的设定语法一定要符合 XML 文件的规则, 如: 字母大小写有差异、标签必须成对, 等等。JSP 2.0 之前, TLD 的文件类型定义的格式是 DTD, 不过 JSP 2.0 之后, 都改用 Schema, 我们先看下列 TLD 起始的声明和定义:

```
<?xml version="1.0" encoding="UTF-8" ?>

<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/
    j2ee web-jsptaglibrary_2_0.xsd"
  version="2.0">
```

从这段叙述中可以知道 Schema 是由 Sun Microsystems 所制定的规范, 而且现在的版本是 2.0。大致了解 TLD 前置的声明、定义之后, 接下来正式开始介绍 Tag Library Descriptor 文件。首先, 笔者把 Tag Library Descriptor 文件分为八个部分: `<taglib>`、`<validator>`、`<listener>`、`<tag>`、`<variable>`、`<attribute>`、`<tag-file>`和`<function>`。

在开始介绍这八个元素之前, 笔者先列出所有元素之间父子关系:

```
<?xml version="1.0" encoding="UTF-8" ?>

<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/
    j2ee web-jsptaglibrary_2_0.xsd"
  version="2.0">
```

```

<tlib-version>1.0</tlib-version>
<description>MyTaglib Powered by javaworld.com.tw</description>
<short-name>Mytaglib</short-name>
*****

<validator>
  <validator-class></validator-class>
  <init-param></init-param>
  *****
</validator>

<listener>
  <listener-class></listener-class>
  *****
</listener>

<tag>
  <name></name>
  <tag-class></tag-class>
  *****

  <variable>
    <name-given></name-given>
    <variable-class></variable-class>
    *****
  </variable>
  *****

  <attribute>
    <name></name>
    <required></required>
    *****
  </attribute>
  *****
</tag>

<tag-file>
  <name></name>
  <path></path>
</tag-file>

<function>
  <description> </description>
  <name> </name>
  <function-class></function-class>
  <function-signature></function-signature>
  *****
</function>
*****
<taglib>

```

图 16-10 为 TLD Schema 元素的结构图，其中星号(*)代表此元素可以没有，也可以有一个或很多个。

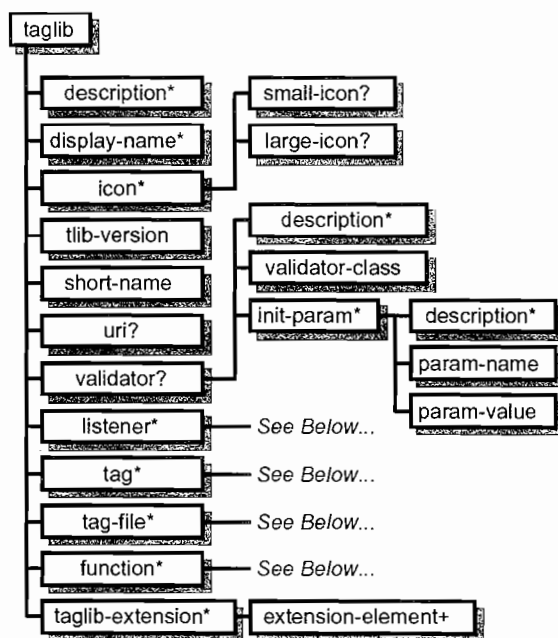


图 16-10 TLD Schema 元素结构图

注意

图 16-10 是从 JSP 2.0 规范书中直接复制下来的。其中有一个地方可能会造成读者的误解，taglib 并无<icon>元素，而只有<small-icon>和<large-icon>，即：

```
<small-icon></small-icon>
<large-icon></large-icon>
```

并不是，如下：

```
<icon>
  <small-icon></small-icon>
  <large-icon></large-icon>
</icon>
```

16-4-1 taglib 元素

<taglib>主要有下列十二项子元素，它们分别为：

- **<tlib-version>**

此元素为必要元素，代表标签函数库的版本。

例如：<tlib-version>1.0</tlib-version>

- **<short-name>**

此元素为必要元素，代表定义标签函数库 taglib 指令中的默认 prefix 值。

例如: `<short-name>Mytaglib</short-name>`

● **<uri>**

用来说明标签函数库的文件来源, 不过通常都是空的。

例如: `<uri></uri>`

● **<description>**

用来叙述说明此标签函数库的相关信息。

例如: `<description>MyTaglib Powered by javaworld.com.tw</description>`

● **<display-name>**

图形化开发工具显示`<display-name>`所指定的名称。

例如: `<display-name>MyTaglib</display-name>`

● **<small-icon>**

在图形化开发工具显示`<small-icon>`所指定的 TLD 相对路径之一 GIF 或 JPEG 的小图标, 大小为 16×16。

例如: `<small-icon>small.gif</small-icon>`

● **<large-icon>**

在图形化开发工具显示`<large-icon>`所指定的 TLD 相对路径之一 GIF 或 JPEG 的大图标, 大小为 32×32。

例如: `<large-icon>large.gif</large-icon>`

● **<validator>**

请参见 16-4-2 小节。

● **<listener>**

请参见 16-4-3 小节。

● **<tag>**

请参见 16-4-4 小节。

● **<tag-file>**

请参见 16-4-7 小节。

● **<function>**

请参见 16-4-8 小节。

一个 taglib 元素的范例, 如下所示:

```
<taglib>
  <tlib-version>1.0</tlib-version>
  <short-name>Mytaglib</short-name>
  <uri></uri>
  <description>
    My Taglib Powered by www.javaworld.com.tw
  </description>
</taglib>
```

```
</description>
<display-name>My Taglib</display-name>
<small-icon>small.gif</small-icon>
<large-icon>large.gif</large-icon>
.....
</taglib>
```

16-4-2 validator 元素

<validator>主要有下列三项子元素，它们分别为：

- **<validator-class>**

设定用户所实现的标签函数库验证类，此类必须继承自 `javax.servlet.jsp.tagext.TagLibraryValidator`。

例如：`<validator-class>tw.com.javaworld.NestingTLV</validator-class>`

- **<init-param>**

可用来初始化验证类所需要的属性，这些属性值会传至验证类。此元素还有三个子元素，分别为：`<param-name>`、`<param-value>`和`<description>`。其中`<param-name>`和`<param-value>`为必要元素，代表属性名称和值，而`<description>`元素用来说明初始化属性。

例如：`<init-name>`

```
    <param-name>inner</param-name>
    <param-value>index</param-value>
</init-name>
```

- **<description>**

用来说明验证类的功能。

例如：`<description>Validates nesting of inner and outer tags.</description>`

16-4-3 listener 元素

<listener>主要有下列五项子元素，它们分别为：

- **<listener-class>**

此元素为必要元素。用来设定用户所实现的事件监听类，此类必须实现下列其中之一接口：

- `javax.servlet.ServletContextListener`
- `javax.servlet.ServletContextAttributeListener`
- `javax.servlet.ServletRequestListener`
- `javax.servlet.ServletRequestAttributeListener`
- `javax.servlet.HttpSessionListener`
- `javax.servlet.HttpSessionAttributeListener`

- **<description>**

用来叙述说明此 Listener 的相关信息。

例如: `<description> MyListener Powered by javaworld.com.tw</description>`

- **<display-name>**

图形化开发工具显示<display-name>所指定的名称。

例如: `<display-name>MyListener</display-name>`

- **<small-icon>**

在图形化开发工具显示<small-icon>所指定的 TLD 相对路径之一 GIF 或 JPEG 的小图标, 大小为 16×16

例如: `<small-icon>small.gif</small-icon>`

- **<large-icon>**

在图形化开发工具显示<large-icon>所指定的 TLD 相对路径之一 GIF 或 JPEG 的大图标, 大小为 32×32

例如: `<large-icon>large.gif</large-icon>`

16-4-4 tag 元素

<tag>有下列十二项子元素, 它主要用来设定用户所自定义的标签。

- **<name>**

用户自定义 Tag 的名称

例如: `<name>Mytag</name>`

- **<tag-class>**

Tag Handler Class 名称

例如: `<tag-class>taglib.Mytag</tag-class>`

- **<tei-class>**

TagExtraInfo 类名称

例如: `<tei-class>taglib.TEI</tei-class>`

- **<body-content>**

它可能的值有四种, 分别为: empty、JSP、scriptless 和 tagdependent。empty 表示标签中没有本体内容。JSP 表示标签中的本体内容可以加入 JSP 的程序代码。scriptless 表示标签中的本体内容可以为 EL、JSP action 元素, 但是不可为 JSP scripting 元素。tagdependent 表示标签中的本体内容交由 tag 自己去处理, 例如:

```
<Mytag:sql >
    SELECT * FROM MemberDB WHERE MemberID < 10
</Mytag:sql >
```

- **<description>**

用来叙述说明此 Tag 的相关信息。

例如: `<description>MyTag Powered by javaworld.com.tw</description>`

- **<display-name>**

图形化开发工具显示<display-name>所指定的名称。

例如: <display-name>MyTag</display-name>

● <small-icon>

在图形化开发工具显示<small-icon>所指定的 TLD 相对路径之一 GIF 或 JPEG 的小图标,大小为 16×16。

例如: <small-icon>small.gif</small-icon>

● <large-icon>

在图形化开发工具显示<large-icon>所指定的 TLD 相对路径之一 GIF 或 JPEG 的大图标,大小为 32×32。

例如: <large-icon>large.gif</large-icon>

● <dynamic-attributes>

此标签是否支持动态属性的功能(必须实现 DynamicAttribute 接口)。

例如: <dynamic-attributes>true</dynamic-attributes>

● <example>

用来增加更多的标签使用说明,包括标签应用时的范例。

● <variable>

请参见 16-4-5 小节。

● <attribute>

请参见 16-4-6 小节。

图 16-11 为 TLD Schema tag 元素的结构图,其中星号(*)代表此元素可有可无。

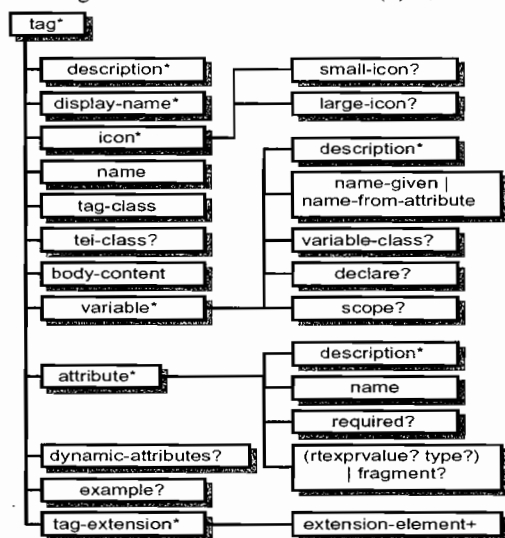


图 16-11 tag 元素的结构图

16-4-5 variable 元素

<variable>有下列五项子元素。它主要用来设定自定义标签的变量。

- **<name-given> | <name-from-attribute>**

<name-given>表示直接指定变量的名称；**<name-from-attribute>**表示以自定义标签的某个属性值为变量名称。15-3-4 小节有更详细的说明。

例如：`<name-given>count</name-given>`

- **<variable-class>**

变量的类型，默认为 `java.lang.String`。

例如：`<variable-class>java.lang.String</variable-class>`

- **<declare>**

代表此变量是否需要声明，默认为 `true`。

例如：`<declare>true</declare>`

- **<scope>**

变量的范围，如：`NESTED`、`AT_BEGIN` 和 `AT_END`，默认为 `NESTED`。

例如：`<scope>AT_END</scope>`

- **<description>**

用来叙述说明此变量的相关信息

16-4-6 attribute 元素

<attribute>有下列六项子元素，它主要用来设定标签的属性。

- **<name>**

属性名称，例如：`<name>num</name>`

- **<required>**

属性是否一定要存在，默认值为 `false`。例如：`<required>true</required>`

- **<rtexprvalue>**

属性值是否可以为 `run-time` 表达式。当此设为 `true` 时，表示说属性值可用动态的方式来指定，如：`<Mytag:Handler num= "${param.num}" />`，假若设为 `false` 时，则一定要用静态的方式来指定属性值。

例如：`<rtexprvalue>true</rtexprvalue>`

- **<type>**

若 **<rtexprvalue>** 为 `true` 时，声明其属性的类型。

例如：`<type>java.util.Date</type>`

- **<fragment>**

属性是否为 JspFragment。默认值为 false。

例如: <fragment>true</fragment>

● **<description>**

用来叙述说明此属性的相关信息。

一个<attribute>范例如下所示:

```
<attribute>
  <name>num</name>
  <required>true</required>
  <rtexprvalue>true</rtexprvalue>
  <type>int</type>
</attribute>
```

16-4-7 tag-file 元素

<tag-file>主要有下列七项子元素, 它们分别为:

● **<name>**

Tag File 的名称。

例如: <name>copyright</name>

● **<path>**

Tag File 的路径。

例如: <path>/META-INF/tags/copyright.tag</path>

● **<description>**

用来叙述说明此 tag-file 的相关信息。

例如: <description> MyTagFile Powered by javaworld.com.tw</description>

● **<display-name>**

图形化开发工具显示<display-name>所指定的名称。

例如: <display-name>MyTag</display-name>

● **<small-icon>**

在图形化开发工具显示<small-icon>所指定的 TLD 相对路径之一 GIF 或 JPEG 的小图标, 大小为 16×16。

例如: <small-icon>small.gif</small-icon>

● **<large-icon>**

在图形化开发工具显示<large-icon>所指定的 TLD 相对路径之一 GIF 或 JPEG 的大图标, 大小为 32×32。

例如: <large-icon>large.gif</large-icon>

● **<example>**

用来增加更多的标签使用说明, 包括标签应用时的范例。

图 16-12 为 TLD Schema tag-file 元素的结构图。

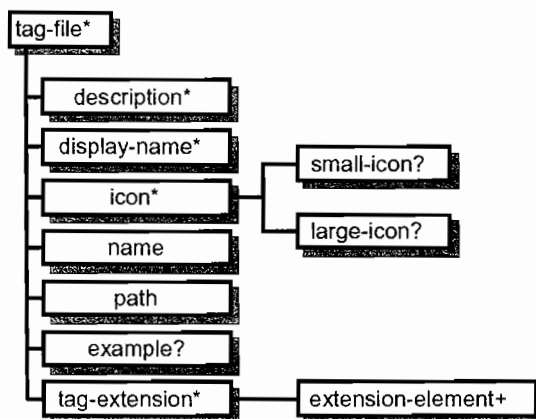


图 16-12 tag-file 元素的结构图

16-4-8 function 元素

<function>主要有下列七项子元素，它们分别为：

- **<name>**

设定 EL 函数的名称。

例如：<name>countVowels</name>

- **<function-class>**

设定 EL 函数的 Java 类。

例如：<function-class>jsp2.examples.el.Functions</function-class>

- **<function-signature>**

设定 EL 函数的传入值和回传值。

例如：<function-signature>
 java.lang.String numVowels(java.lang.String)
 </function-signature>

- **<description>**

用来叙述说明此 function 的相关信息。

例如：<description>
 Counts the number of vowels (a,e,i,o,u) in the given String
 </description>

- **<display-name>**

图形化开发工具显示<display-name>所指定的名称。

例如: <description>MyListener</description>

● <small-icon>

在图形化开发工具显示<small-icon>所指定的 TLD 相对路径之一 GIF 或 JPEG 的小图标, 大小为 16×16。

例如: <small-icon>small.gif</small-icon >

● <large-icon>

在图形化开发工具显示<large-icon>所指定的 TLD 相对路径之一 GIF 或 JPEG 的大图标, 大小为 32×32。

例如: <large-icon>large.gif</large-icon >

● <example>

用来增加更多的标签使用说明, 包括标签应用时的范例。

图 16-13 为 TLD Schema function 元素的结构图。

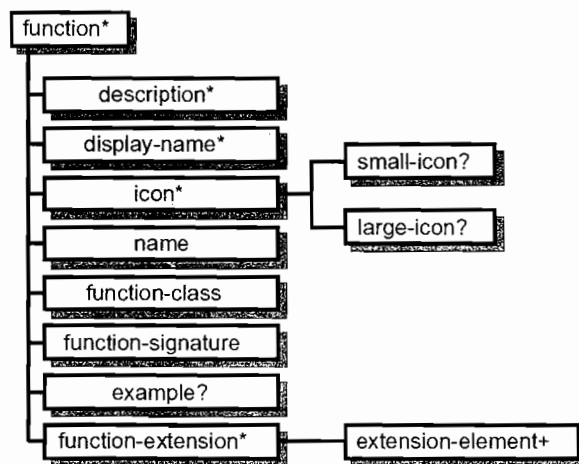


图 16-13 function 元素的结构图

17

第十七章

JSP 与 JavaMail

笔者将在本章介绍如何利用 JavaMail API 搭配 JSP 来发送电子邮件。电子邮件是目前网络上使用最多的服务，笔者举几个使用电子邮件的例子：如果用户在网站是第一次注册，为了安全保密起见，则会将密码用电子邮件的方式交给用户。一方面网站管理人员能够确定用户的电子信箱是合法的；另一方面用户也能得到保障。假若用户忘记密码时，用户也只要在网站上填入自己的电子邮件地址，然后系统再自动把密码用电子邮件寄送给用户，这对双方来说，都是最方便，也较有保障的方式。

本章将分 5 节来介绍如何使用 JavaMail API 来传送电子邮件：

- 17-1 JavaMail 1.3.1 的介绍与使用方法
- 17-2 JavaMail 范例程序一——传送一般邮件
- 17-3 JavaMail 范例程序二——传送 HTML 格式的邮件
- 17-4 JavaMail 范例程序三——传送附件
- 17-5 JavaMail 范例程序四——传送自定义内容的邮件

JSP2.0 技术手册

17-1 JavaMail 1.3.1 的介绍与使用方法

对于 Java 平台而言, JavaMail 算是比较晚加入的 API, 目前在 Java 网站 (<http://www.javasoft.com/products/javamail/>) 上最新的版本是 JavaMail 1.3.1 API, 我们直接使用它来作为开发工具。

JavaMail 是从某些现存的信息系统里抽取出来 (例如: IMAP、POP3 等等) 的, 它的表现可以说是非常的稳固。JavaMail API 定义了一个用来管理邮件的共通接口, 并且 JavaMail 允许程序员通过 API 里的接口来撰写自己的应用程序, 然后在执行时期再请求使用某种类型的处理。

在 JavaMail API 中定义四个主要组件: Message、Folder、Store 和 Session。

- **Message:** Message 类定义邮件信息之内容类型等的属性, 所有的信息都储存在文件夹 (Folder) 中, 例如: INDEX 之类的文件夹, 而每个文件夹还可以包含子文件夹, 以便产生类似树状结构的层级。
- **Folder:** Folder 类定义存取(fetch)、备份(copy)、附加(append)以及删除(delete)信息等等的方法。
- **Store:** Store 类定义了用来保存文件夹层级的数据库, 以及包含在文件夹之中的信息, 它也可以定义存取协议的类型, 以便用来存取文件夹与信息。
- **Session:** Session 类负责验证用户, 以及控制信息储藏库(store)与传送(transport)邮件的存取权限。

■ 构建 JavaMail 的开发环境

1. mail.jar

首先读者可以至 <http://www.javasoft.com/products/javamail/> 下载最新的 JavaMail API 或是直接取用光盘中的 JavaMail 1.3.1 API, 文件名为 *javamail-1_3_1.zip*。首先将 *javamail-1_3_1.zip* 解压缩后, *mail.jar* 是最重要的文件, 它里面就是 JavaMail API 的类, 除了 *mail.jar* 之外, 它还内附 *imap.jar*、*pop3.jar* 和 *smtp.jar*。

2. activation.jar

除了 *mail.jar* 之外, 读者还可以至 <http://java.sun.com/beans/glasgow/jaf.html> 下载最新的 JavaBeans Activation Framework(JAF), 目前最新版本为 1.0.2, 光盘中也有为各位读者收录。

这些前置工作都准备好后, 接下来就直接看范例程序, 相信读者很快就能知道如何使用 JavaMail 来发信。

17-2 JavaMail 范例程序——传送一般邮件

这节将写一个最简单的发信程序。我们先做一个 *JavaMail.html* 的用户接口(如图 17-1 所示), 让用户输入数据, 然后再传送到 *JavaMail.jsp* 做处理。

■ *JavaMail.html*

```
<html>
<head>
  <title>CH17 - JavaMail.html</title>
  <meta http-equiv="Content-Type" content="text/html; charset=GB2312">
</head>
<body>

<h2>利用 JavaMail 来发送电子邮件</h2>
<form name="Form" method="post" action="JavaMail.jsp">
  <p>寄信人: <input type="text" name="From" size="30" maxlength="30"></p>
  <p>收信人: <input type="text" name="To" size="30" maxlength="30"></p>
  <p>主题: <input type="text" name="Subject" size="30" maxlength="30"></p>
  <p>内容: </p><p><textarea name="Message" cols=40 rows=5></textarea></p>

  <input type="submit" value="发送">
  <input type="reset" value="清除">
</form>

</body>
</html>
```

JavaMail.html 的执行结果如图 17-1 所示: 其中包括寄信人、收信人、主题和内容。当用户填好数据后, 按下【发送】, 就会将窗体传送到 *JavaMail.jsp* 做发送电子邮件的工作。

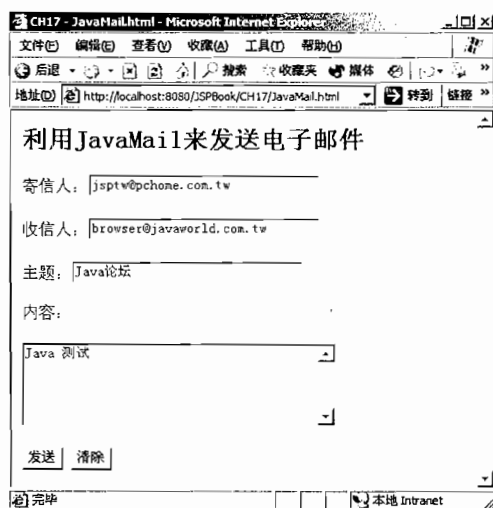


图 17-1 *JavaMail.html* 的执行结果

■ JavaMail.jsp

```
<%@ page import="javax.mail.*" %>
<%@ page import="javax.mail.internet.*" %>
<%@ page import="javax.activation.*" %>
<%@ page import="java.util.*,java.io.*" %>
<%@ page contentType="text/html;charset=GB2312" %>

<html>
<head>
  <title>CH17 - JavaMail.jsp</title>
</head>
<body>

<h2>利用 JavaMail 来传递电子邮件 </h2>
<%

    InetAddress[] address = null;
    request.setCharacterEncoding("GB2312");

    String mailserver    = "ecd.pchome.com.tw";
    String From          = request.getParameter("From");
    String to            = request.getParameter("To");
    String Subject       = request.getParameter("Subject");
    String messageText   = request.getParameter("Message");

    boolean sessionDebug = false;

try {

    // 设定所要用的 Mail 服务器和所使用的传输协议
    java.util.Properties props = System.getProperties();
    props.put("mail.host",mailserver);
    props.put("mail.transport.protocol","smtp");

    // 产生新的 Session 服务
    javax.mail.Session mailSession =
        javax.mail.Session.getDefaultInstance(props,null);
    mailSession.setDebug(sessionDebug);

    Message msg = new MimeMessage(mailSession);

    // 设定传递邮件的发信人
    msg.setFrom(new InetAddress(From));

    // 设定传递邮件至收信人的信箱
    address = InetAddress.parse(to,false);
    msg.setRecipients(Message.RecipientType.TO, address);

    // 设定信中的主题
    msg.setSubject(Subject);
    // 设定送信的时间
```

JSP2.0 技术手册


```

msg.setSentDate(new Date());

// 设定传送信的 MIME Type
msg.setText(messageText);

// 送信
Transport.send(msg);

out.println("邮件已顺利传送");
}
catch (MessagingException mex) {
    mex.printStackTrace();
}
%>

</body>
</html>

```

JavaMail.jsp 范例中, 笔者以 PChome 的邮件服务器来做测试, 因此当读者在执行这个范例程序时, 你的寄信人在填写时, 必须为 **xxx@pchome.com.tw**, 不然就不能传送邮件。你也可以使用其他免费电子邮件信箱, 只要它有提供 POP3 的服务, 所以 yahoo 的免费电子邮件就不能拿来测试。

首先, 我们必须导入 `javax.mail.*`、`javax.mail.internet.*` 和 `javax.activation.*` 这三个套件, 前面两项在 *mail.jar* 中, 最后一项在 *activation.jar*。所以, 记得将这三个套件加到 Container 之中, 如下所示:

```

<%@ page import="javax.mail.*" %>
<%@ page import="javax.mail.internet.*" %>
<%@ page import="javax.activation.*" %>

```

再来就是设定 Mail 服务器和传输协议, 设定的方法如下:

```

String mailserver = "msk.pchome.com.tw";

.....
java.util.Properties props = System.getProperties();
props.put("mail.host", mailserver);
props.put("mail.transport.protocol", "smtp");

```

这个范例中, 笔者默认的 Mail 服务器是 PChome 的邮件服务器, 然后使用的传输协议是 SMTP。再就是一些固定的标准动作, 如: 产生 Mail Session、设定寄信人、收信人的邮件地址, 如下所示:

```

javax.mail.Session mailSession =
    javax.mail.Session.getDefaultInstance(props, null);
mailSession.setDebug(sessionDebug);

Message msg = new MimeMessage(mailSession);

// 设定邮件的发信人

```

JSP2.0 技术手册

```

msg.setFrom(new InternetAddress(From));

// 设定邮件收信人的信箱
address = InternetAddress.parse(to, false);
msg.setRecipients(Message.RecipientType.TO, address);

```

再来就是要设定有关邮件本身，例如：邮件的主题、内容，还有 MIME 类型等等，设定方法如下：

```

// 设定邮件的主题
msg.setSubject(Subject);
// 设定发信的时间
msg.setSentDate(new Date());

// 设定邮件的 MIME Type
msg.setText(messageText);

```

最后再调用

```
Transport.send(msg);
```

传送我们的电子邮件。

这个范例是最简单的程序，接下来要介绍如何传送 HTML 格式的电子邮件，并且能够附加文件，请各位读者往下看这个 JavaMail 的范例程序。

17-3 JavaMail 范例程序二——传送 HTML 格式的邮件

目前许多网站也推出电子期刊的功能，不过为了更能够吸引用户来阅读，通常电子期刊的内容也摆脱以往单调的文本文件格式，改用 HTML 的格式，希望利用 HTML 来传送多姿多彩的电子期刊内容。所以，我们将之前的范例程序再稍做修改，新增传送 Html 格式邮件的功能，就成了 *JavaMail2.html* (如图 17-2 所示) 和 *JavaMail2.jsp*。

■ *JavaMail2.html*

```

<html>
<head>
  <title>CH17 - JavaMail2.html</title>
  <meta http-equiv="Content-Type" content="text/html; charset=GB2312">
</head>
<body>

<h2>利用 JavaMail 来传送电子邮件 - HTML 格式</h2>
<form name="SendMessage" Method="post" action="JavaMail2.jsp" >
  <p>寄信人: <input type="text" name="From" size="30" maxlength="30"></p>
  <p>收信人: <input type="text" name="To" size="30" maxlength="30"></p>
  <p>主题: <input type="text" name="Subject" size="30" maxlength="30"></p>
  <p>格式: <select name="Type" size="1">
    <option value="text/plain">Text</option>

```

JSP2.0 技术手册

```

        <option value="text/html">HTML</option>
    </select></p>
    <p>内容: </p><p><textarea name="Message" cols=40 rows=5></textarea></p>
    <input type="submit" value="传送">
    <input type="reset" value="清除">
</form>
</body>
</html>

```

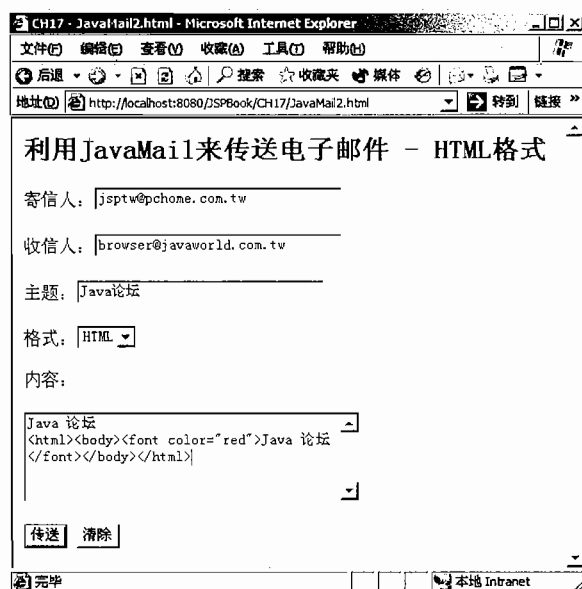


图 17-2 JavaMail2.html 的执行结果

■ JavaMail2.jsp

```

<%@ page import="javax.mail.*" %>
<%@ page import="javax.activation.*" %>
<%@ page import="javax.mail.internet.*" %>
<%@ page import="java.util.*,java.io.*" %>

<%@ page contentType="text/html; charset=GB2312" %>

<html>
<head>
    <title>CH17 - JavaMail2.jsp</title>
</head>
<body>

<h2>利用 JavaMail 来传送电子邮件 - HTML 格式</h2>

```

```
<%
    InetAddress[] address = null;

    request.setCharacterEncoding("GB2312");

    String mailserver = "ecd.pchome.com.tw";
    String From = request.getParameter("From");
    String to = request.getParameter("To");
    String Subject = request.getParameter("Subject");
    String type = request.getParameter("Type");
    String messageText = request.getParameter("Message");

    boolean sessionDebug = false;

    try {

        // 设定所要用的 Mail 服务器和所使用的传输协议
        java.util.Properties props = System.getProperties();

        props.put("mail.host", mailserver);
        props.put("mail.transport.protocol", "smtp");

        // 产生新的 Session 服务
        javax.mail.Session mailSession =
            javax.mail.Session.getDefaultInstance(props, null);
        mailSession.setDebug(sessionDebug);

        Message msg = new MimeMessage(mailSession);

        // 设定传送邮件的发信人
        msg.setFrom(new InetAddress(From));

        // 设定传送邮件至收信人的信箱
        address = InetAddress.parse(to, false);
        msg.setRecipients(Message.RecipientType.TO, address);

        // 设定信中的主题
        msg.setSubject(Subject);

        // 设定送信的时间
        msg.setSentDate(new Date());

        Multipart mp = new MimeMultipart();
        MimeBodyPart mbp = new MimeBodyPart();

        // 设定邮件内容的类型为 text/plain 或 text/html
        mbp.setContent(messageText, type + "; charset=GB2312");
        mp.addBodyPart(mbp);
        msg.setContent(mp);

        Transport.send(msg);
    }
}
```

JSP2.0 技术手册

```

        out.println("邮件已顺利发送");
    }
    catch (MessagingException mex)
    {
        mex.printStackTrace();
    }
%>

</body>
</html>

```

JavaMail2.jsp 当中, 为了能传送 HTML 格式的邮件功能, 我们必须利用 `MimeBodyPart` 类提供 `setContent()` 的方法来设定邮件的 MIME 类型。假若要传送 HTML 格式的邮件, 只需要把 `setContent()` 第二个参数设为 `text/html`, 即可达到传送 HTML 格式邮件的功能, 如下所示:

```
msg.setContent(messageText, "text/html");
```

假若在邮件内容中的中文变为乱码时, 则可以设定它的编码方式为 GB2312, 如下所示:

```
msg.setContent(messageText.toString(), "text/html; charset=GB2312");
```

其余部分在程序代码中皆有批注, 希望这个范例程序能够解决读者在开发网站时遇到和 JavaMail 有关的问题。*JavaMail2.jsp* 执行后, 笔者使用 Microsoft Outlook 收信的结果如图 17-3:

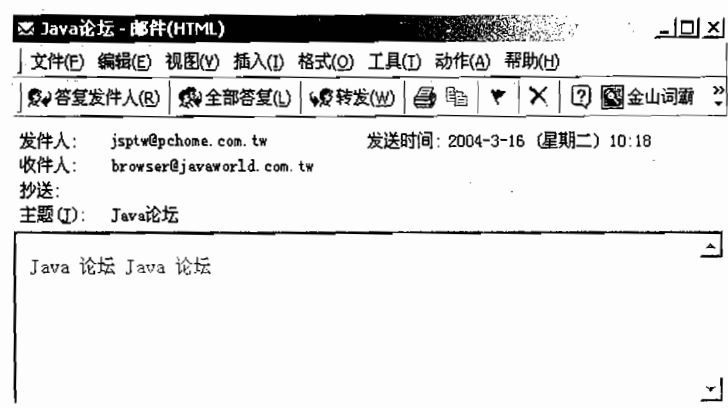


图 17-3 *JavaMail2.jsp* 执行后收信的画面

17-4 JavaMail 范例程序三——传送附件

看完上一个范例之后, 已经学会如何传送 HTML 格式的电子邮件, 接下来的范例程序新增附件的功能。附件的功能需要用到上传文件的机制, 因此笔者使用欧莱礼的上传套件, 假若读者不熟悉上传文件的功能时, 可以先阅读“第九章: 网页窗体的处理”。本章的范例程序, 一样有两个文件, 分别为 *JavaMail3.html* 和 *JavaMail3.jsp*。

■ JavaMail3.html

```
<html>
<head>
  <title>CH17 - JavaMail3.html</title>
  <meta http-equiv="Content-Type" content="text/html; charset=GB2312">
</head>
<body>

<h2>利用 JavaMail 来传送电子邮件 - 附件</h2>
<form name="SendMessage" Method="post" action="JavaMail3.jsp"
enctype="multipart/form-data">
  <p>寄信人: <input type="text" name="From" size="30" maxlength="30"></p>
  <p>收信人: <input type="text" name="To" size="30" maxlength="30"></p>
  <p>主题: <input type="text" name="Subject" size="30" maxlength="30"></p>
  <p>格式: <select name="Type" size="1">
    <option value="text/plain">Text</option>
    <option value="text/html">HTML</option>
  </select></p>
  <p>附件: <input type="file" name="FileName" size="20" maxlength="20"></p>
  <p>内容: </p><p><textarea name="Message" cols=40 rows=5></textarea></p>

  <input type="submit" value="传送">
  <input type="reset" value="清除">
</form>

</body>
</html>
```

JavaMail3.html 的执行结果如图 17-4:

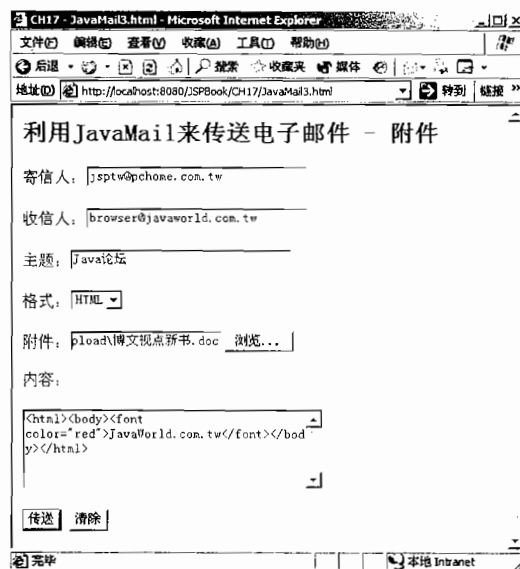


图 17-4 JavaMail3.html 的执行结果

接下来我们看 `JavaMail3.jsp` 如何传送附件的电子邮件。

■ `JavaMail3.jsp`

```
<%@ page import="javax.mail.*" %>
<%@ page import="javax.mail.internet.*" %>
<%@ page import="javax.activation.*" %>
<%@ page import="java.util.*,java.io.*" %>
<%@ page import="com.oreilly.servlet.MultipartRequest" %>

<%@ page contentType="text/html; charset=GB2312" %>

<html>
<head>
<title>CH17 - JavaMail2.jsp</title>
</head>
<body>

<h2>利用 JavaMail 来传送电子邮件 -附件</h2>

<%
    InetAddress[] address = null;

    request.setCharacterEncoding("GB2312");
    MultipartRequest multi = new MultipartRequest(request, ".", 5*1024*1024,
                                                "GB2312");

    String mailserver = "ecd.pchome.com.tw";
    String From       = multi.getParameter("From");
    String to         = multi.getParameter("To");
    String Subject    = multi.getParameter("Subject");
    String type       = multi.getParameter("Type");
    String messageText = multi.getParameter("Message");
    String FileName   = multi.getFilesystemName("FileName");

    boolean sessionDebug = false;

    try {

        // 设定所要用的 Mail 服务器和所使用的传输协议
        java.util.Properties props = System.getProperties();

        props.put("mail.host", mailserver);
        props.put("mail.transport.protocol", "smtp");

        // 产生新的 Session 服务
        javax.mail.Session mailSession =
            javax.mail.Session.getDefaultInstance(props, null);
        mailSession.setDebug(sessionDebug);

        Message msg = new MimeMessage(mailSession);

        // 设定传送邮件的发信人
        msg.setFrom(new InetAddress(From));
```

JSP2.0 技术手册

```

// 设定传送邮件至收信人的信箱
address = InternetAddress.parse(to, false);
msg.setRecipients(Message.RecipientType.TO, address);

// 设定信中的主题
msg.setSubject(Subject);

// 设定送信的时间
msg.setSentDate(new Date());

if (FileName != null)
{
    File file = new File(FileName);

    // 如果有附件, 先将邮件内容部分存起来
    MimeBodyPart mbp1 = new MimeBodyPart();

    // 设定邮件内容的类型为 text/plain 或 text/html
    mbp1.setContent(messageText, type + ";charset=GB2312");

    // 再来对附件做处理
    MimeBodyPart mbp2 = new MimeBodyPart();
    FileDataSource fds = new FileDataSource(FileName);
    mbp2.setDataHandler(new DataHandler(fds));
    mbp2.setFileName(MimeUtility.encodeText(fds.getName(),
        "GB2312", "B"));

    // 最后再将两者集成起来, 当做一份邮件送出
    Multipart mp = new MimeMultipart();
    mp.addBodyPart(mbp1);
    mp.addBodyPart(mbp2);
    msg.setContent(mp);
}
else
{
    // 若没有附件, 就直接存邮件内容
    msg.setContent(messageText, type + ";charset=GB2312");
}

Transport.send(msg);
out.println("邮件已顺利发送");
}
catch (MessagingException mex)
{
    mex.printStackTrace();
}
}
%>

</body>
</html>

```

如何把文件连同邮件一起传送到收信人。下列为处理邮件附件的部分程序代码:

```
// 如果有附件, 先将邮件内容部分存起来
```

JSP2.0 技术手册

```

MimeBodyPart mbp1 = new MimeBodyPart();

// 设定邮件内容的类型为 text/plain 或 text/html
mbp1.setContent(messageText, type + ";charset=GB2312");

// 再来对附件做处理
MimeBodyPart mbp2 = new MimeBodyPart();
FileDataSource fds = new FileDataSource(FileName);
mbp2.setDataHandler(new DataHandler(fds));
mbp2.setFileName(MimeUtility.encodeText(fds.getName(), "GB2312", "B"));

// 最后再将两者集成起来, 当做一份邮件送出
Multipart mp = new Multipart();
mp.addBodyPart(mbp1);
mp.addBodyPart(mbp2);
msg.setContent(mp);

```

附件的做法是：先将处理好的邮件内容存入到 `MimeBodyPart` 对象中；然后将文件存入到另一个 `MimeBodyPart` 对象中；最后把两个 `MimeBodyPart` 对象(一份是邮件内容本身，另一份是文件部分)通通存入到 `Multipart` 对象，合而为一加入 `Message` 对象中，传送到收信人的 Mail 服务器。为了解决附加中文文件的问题，我们必须加上一段程序来做转码的工作，如下：

```
mbp2.setFileName(MimeUtility.encodeText(fds.getName(), "GB2312", "B"));
```

`JavaMail3.jsp` 执行后，笔者使用 Microsoft Outlook 收信的结果如图 17-5：

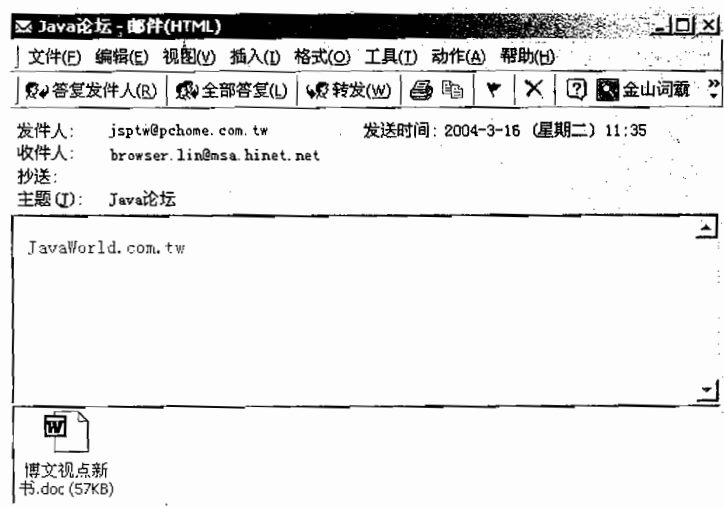


图 17-5 `JavaMail3.jsp` 执行后，收信的画面

17-5 JavaMail 范例程序四——传送自定义内容的邮件

相信大家曾经有过这样的经验：有些网站会先要求用户注册后才能执行网站内的某些功能。

JSP2.0 技术手册

当注册成功之后，网站会自动发一封确认的信件给注册的用户。对网站来说，每一封确认邮件的内容都相同，惟一不同的只有收信人的名称、密码等等。

为了达成上述的功能，通常我们会把邮件的内容先制作成一个模板(template)，它的内容包含两部分，其中一部分是固定不变的信息；另一部分就是变动的信息，例如：

```
<html>
<body>
  Hi, <b>{name}</b><br>
  Your Password is : {password}<br>
  Welcome to <a href="http://www.jsptw.com/jute">www.javaworld.com.tw</a>
</body>
</html>
```

这一封 HTML 格式的邮件模板，大部分的内容都是一些固定不变的信息，只有{name}和{password}才代表变量，其中{name}表示用户的名称；{password}表示用户的密码。

一般直观的做法，可能会写一个程序来剖析这份模板文件，把{name}和{password}取代成所需的值。不过这样的做法有两个很大的缺点：

- (1) 繁杂费时：因为我们必须要自己写一个剖析程序来取代模板中的变量。
- (2) 缺乏弹性：当模板有变动时，同时也要更新剖析程序。

因此，笔者采用 Java 提供的 MessageFormat 类来完成这项功能。接下来介绍一个完整的范例，这个范例主要有三个部分：Mail.properties、JavaMail4.html 和 JavaMail4.jsp。

首先，Mail.properties 就是范例中的邮件模板，它的内容如下：

■ Mail.properties

```
message = <html> \
  <body> \
    Hi, <b>{0}</b><br> \
    Your Password is : {1}<br> \
    Welcome to <a href = \
"www.javaworld.com.tw">www.javaworld.com.tw</a> \
  </body> \
</html> \
```

Mail.properties 中，message 等号右边的内容就是邮件内容。笔者将发送一封 HTML 格式的邮件，它有两个变量：{0}和{1}。其中{0}代表收件人的名称；{1}代表收件人的密码。

注意

.properties 文件的格式是依照 key = value 的方式，因此 Mail.properties 应该为：
 message = <html><body>Hi, {0}
 Your Password is : {1}

 Welcome to www.javaworld.com.tw
 </body></html>

不过为了让程序更容易阅读，笔者在需要换行的最后，加上 \。

Mail.properties 必须放在 WEB-INF/classes 目录下，然后 ResourceBundle 才能顺利读取

Mail.properties 的内容。

```
ResourceBundle messages = ResourceBundle.getBundle("Mail");
```

接下来看 *JavaMail4.html* 的内容：

■ *JavaMail4.html*

```
<html>
<head>
  <title>CH17 - JavaMail4.html</title>
  <meta http-equiv="Content-Type" content="text/html; charset=GB2312">
</head>
<body>

<h2>利用 JavaMail 来传送注册信息</h2>
<form name="Form" method="post" action="JavaMail4.jsp">
  <p>用户名称: <input type="text" name="Name" size="30" ></p>
  <p>用户密码: <input type="password" name="Password" size="30" ></p>
  <p>用户 E-mail: <input type="text" name="To" size="30" ></p>

  <input type="submit" value="注册">
  <input type="reset" value="清除">
</form>

</body>
</html>
```

JavaMail4.html 有三个字段，其中用户名称和密码就是 *Mail.properties* 中两个变量所代表的值。最后一个字段“用户 E-mail”就是收件人的 E-mail。*JavaMail4.html* 的执行结果如图 17-6：

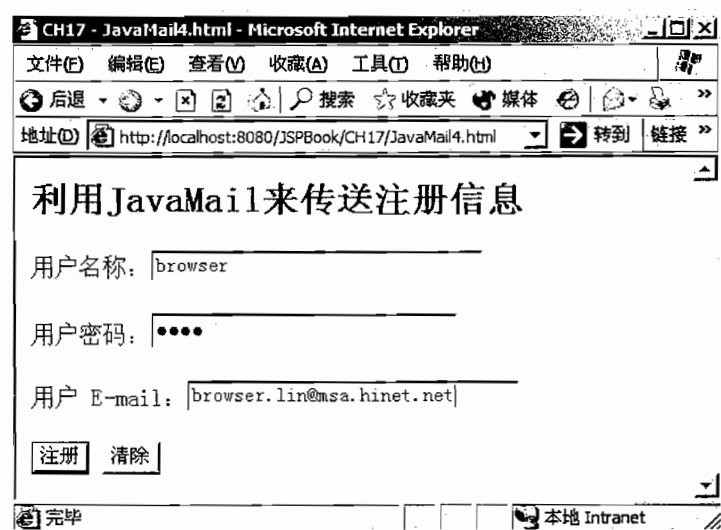


图 17-6 *JavaMail4.html* 的执行结果

最后来看主要的程序 *JavaMail4.jsp*:

■ *JavaMail4.jsp*

```
<%@ page import="javax.mail.*" %>
<%@ page import="javax.activation.*" %>
<%@ page import="javax.mail.internet.*" %>
<%@ page import="java.util.*,java.io.*,java.text.*" %>

<%@ page contentType="text/html; charset=GB2312" %>

<html>
<head>
  <title>CH17 - JavaMail4.jsp</title>
</head>
<body>

<h2>利用 JavaMail 来传送注册信息</h2>
<%
  InetAddress[] address = null;
  ResourceBundle messages = ResourceBundle.getBundle("Mail");

  request.setCharacterEncoding("GB2312");

  String mailserver = "ecd.pchome.com.tw";
  String From       = "jsptw@pchome.com.tw";
  String Subject    = "欢迎加入 JavaWorld 论坛";
  String name       = request.getParameter("Name");
  String password   = request.getParameter("Password");
  String to         = request.getParameter("To");

  Object[] args = { name , password};
  MessageFormat formatter = new MessageFormat("");
  formatter.applyPattern(messages.getString("message"));
  String messageText = formatter.format(args);

  boolean sessionDebug = false;

  try {

    // 设定所要用的 Mail 服务器和所使用的传输协议
    java.util.Properties props = System.getProperties();

    props.put("mail.host",mailserver);
    props.put("mail.transport.protocol","smtp");

    // 产生新的 Session 服务
    javax.mail.Session mailSession =
      javax.mail.Session.getDefaultInstance(props,null);
    mailSession.setDebug(sessionDebug);

    Message msg = new MimeMessage(mailSession);

    // 设定传送邮件的发信人
```

JSP2.0 技术手册


```

msg.setFrom(new InternetAddress(From));

// 设定传送邮件至收信人的信箱
address = InternetAddress.parse(to, false);
msg.setRecipients(Message.RecipientType.TO, address);

// 设定信中的主题
msg.setSubject(Subject);

// 设定送信的时间
msg.setSentDate(new Date());

Multipart mp = new MimeMultipart();
MimeBodyPart mbp = new MimeBodyPart();

// 设定邮件内容的类型为 text/html
mbp.setContent(messageText, "text/html; charset=GB2312");
mp.addBodyPart(mbp);
msg.setContent(mp);

Transport.send(msg);
out.println("邮件已顺利发送");
}
catch (MessagingException mex)
{
    mex.printStackTrace();
}
%>

```

JavaMail4.jsp 和之前范例最大的区别在于：邮件的内容是读取 *Mail.properties*，并且将变量取代为 *JavaMail4.html* 所输入的用户名称和密码。因此这部分最主要的程序代码如下：

```

<%@ page import=" java.text.*" %>

<%
    ResourceBundle messages = ResourceBundle.getBundle("Mail");

    String mailserver = "ecd.pchome.com.tw";
    String From       = "jsptw@pchome.com.tw";
    String Subject    = "欢迎加入 JavaWorld 论坛";
    String name       = request.getParameter("Name");
    String password    = request.getParameter("Password");
    String to         = request.getParameter("To");

    Object[] args = { name , password};
    MessageFormat formatter = new MessageFormat("");
    formatter.applyPattern(messages.getString("message"));
    String messageText = formatter.format(args);
    .....
%>

```

这个范例主要使用到两个类: `java.util.ResourceBundle` 和 `java.text.MessageFormat`。使用 `ResourceBundle.getBundle()` 将 `Mail.properties` 文件的内容读取进来, 然后再调用 `ResourceBundle.getString("message")` 取得 `message` 等号右边的内容, 即

```
<html><body>Hi, <b>{0}</b><br>Your Password is : {1}<br>Welcome to <a href="www.javaworld.com.tw">www.javaworld.com.tw</a></body></html>
```

取得之后, 再利用 `MessageFormat` 类, 将 `{0}` 和 `{1}` 各自取代成 `name` 和 `password`:

```
Object[] args = { name , password};  
String messageText = formatter.format(args);
```

最后 `messageText` 即为 HTML 格式的邮件内容。`JavaMail4.jsp` 执行后, 笔者使用 Microsoft Outlook 收信的结果如图 17-7:



图 17-7 JavaMail4.jsp 执行后, 使用 Microsoft Outlook 收信的结果

18

第十八章

Web 架构——MVC

本章我们将探讨 Web 的架构，并且介绍使用 JSP 开发 Web 应用程序可使用的模块(model)。本章的重点不在于如何写 JSP 的程序，而是教导各位读者如何规划网站的架构，只要事前适当地规划，日后不论在新增、修改、维护时，都可将繁杂的工作加以简化。本章将分 3 节来介绍：

- 18-1 MVC (Model-View-Controller)的介绍
- 18-2 Model 1 与 Model 2 的介绍
- 18-3 Model 1 与 Model 2 的范例程序

JSP2.0 技术手册

18-1 MVC (Model - View - Controller)的介绍

通常我们可以将整个网站应用程序的架构分为三个部分：

■ 显示层(Presentation Layer)

显示层包括前端的 HTML、XML 和 Applets 等，这层主要当做用户的操作接口，让用户输入数据和显示数据处理后的结果。该层的功能就如同 MVC 中的 View 部分。

显示层无须知道数据是如何取得或是数据该交由谁处理，它只专注于显示数据、结果等等，至于其他部分如：数据的处理、运算等等，皆与该层无关。

■ 商业逻辑层(Business Logic Layer)

这层将是整个网站的核心部分，它的功能包括：数据处理、连接数据库和产生数据。该层就如同 MVC 中的 Model 部分。

商业逻辑层通常是开发人员最需要专注的地方，日后若要维护网站时，只需要在该部分做更新、异动，不再须要牵涉到显示层的部分，造成美工人员的困扰，这样一来，即可将网页设计和程序处理做完整的分离，日后在维护时，弹性能够更大，过程也能简化。

■ 控制层(Control Layer)

这层主要的工作就是控制整个网站处理的流程。它的角色通常是介于显示层和商业逻辑层之中，是 MVC 中的 Controller。

控制层将显示层得到的数据，判断应交由哪个商业逻辑层做处理，然后再将结果交由显示层，显示出数据处理后的结果。如图 18-1 所示。

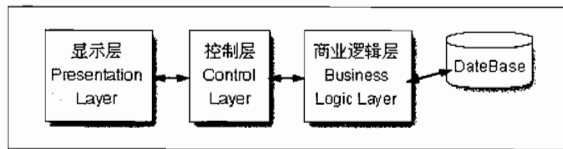


图 18-1 网站架构的三个部分

本节利用一般网站的三层架构来说明 MVC(Model-View-Controller)，相信各位读者已经了解了 MVC 各自扮演的角色和用途。MVC 最主要的精神之一就是 Model 和 View 的分离，它们两者的分离，可使日后网页设计师和程序员能够独立工作，互不影响，从而增加维护的

效率。

除了 Model 和 View 分离之外,将商业逻辑层的数据处理建立成许多的组件,增加程序的可重用性,一方面能减轻程序员的工作负担和减少公司开发成本;另一方面增进网站功能扩充的弹性,这部分是笔者认为最重要的一点。

18-2 Model 1 与 Model 2 的介绍

使用 Java 来开发网站应用程序的时候,通常可以分 Model 1 及 Model 2 两种设计模块。

18-2-1 Model 1

因为 JSP 开发十分简单,对于小型的系统能够马上构建及运作,很快地取代 Servlet 的地位,成为构建网站应用程序的主要语言。在 JSP 网页中很容易结合商业逻辑(`jsp:useBean`)、流程管控(`scriptlet` `<%..%>`)及 HTML (`<html>`)快速地开发出一套系统。有许多的网站就是通过一组 JSP 的结合所制作出来的,因此这种以 JSP 为中心的设计模式,我们就称为 Model 1。

Model 1 的处理方式其实还可以分为两种,一种是完全使用 JSP 来开发,另外一种则是使用 JSP + JavaBean 的设计,下列将分别对于这两种运作模式加以说明。

■ Model 1: 纯粹使用 JSP 开发系统

当用户发出一个请求到服务器端,就是由 JSP 来接收处理,接着将执行结果响应到客户端(见图 18-2)。

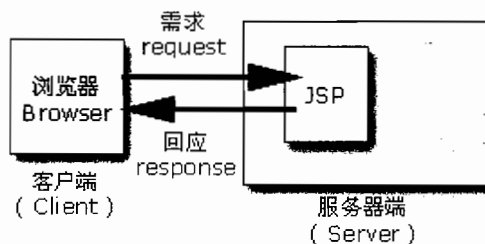


图 18-2 Model 1: 纯粹使用 JSP 开发系统

这种做法的优点为:

- 开发时程缩短: 程序员无须撰写额外的 Servlet 及 JavaBean, 只须专注地开发 JSP。
- 小幅度修改非常容易: 因为没有使用到 Servlet 及 JavaBean, 修改小幅度的程序代码, 无须重新编译, 直接存盘后交由 JSP Container 执行。

但是该方式也有许多的缺点:

- 程序可读性降低: 因为程序代码与网页标签混合在一起, 从而增加维护的困难度。
- 程序重复利用性降低: 因为程序都撰写在 JSP 之中, 往往会在不同的 JSP 中找到相同的程序代码, 当商业逻辑修改的时候, 就必须修改所有相关的 JSP, 造成的负荷也就更大。

■ Model 1: 使用 JSP + JavaBean 开发系统

相对于纯粹使用 JSP 开发网站应用程序, 许多有经验的工程师都会将部分可以重复利用的组件抽出来写成 JavaBean, 当用户送来一个需求时, 通过 JSP 调用 JavaBean 负责相关数据存取、逻辑运算等等的处理, 最后将结果回传到 JSP 显示结果(见图 18-3)。

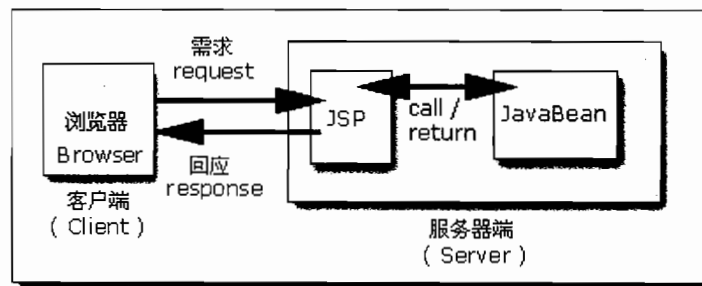


图 18-3 Model 1: 使用 JSP+JavaBean 开发系统

这种做法的优点为:

- 程序可读性增高: 将复杂的程序代码写在 JavaBean 之中, 减少和网页标签混合的情况, 未来维护的时候能够较为轻松。
- 可重复利用性高: 由于通过 JavaBean 来封装重要的商业逻辑运算, 不同的 JSP 可以调用许多共享性的组件, 减少开发重复程序代码的工作, 增加开发效能。

该方式也有一些缺点:

- 缺乏流程控制: 这是 Model 1 最大的缺点, 缺少了 MVC 中的 Controller 去控制相关的流程, 每一个 JSP 都要验证需求的参数正确度、确认用户的身份权限、异常发生的处理, 甚至还包括显示端的网页编码原则及语系设定。

就 Model 1 整体来说, 进行快速及小型的项目的应用开发具有非常大的优势, 但是这样开发的结果却造成未来维护不易的问题, 非常不利于应用程序的扩展与更新, 因此大型系统的开发大多采取 Model 2 MVC 架构的开发模式。

18-2-2 Model 2

以 Java 来开发网站应用程序, 俗称的 Model 2 就是采用 MVC 架构的开发模式。MVC

JSP2.0 技术手册

是 Model-View-Controller 的缩写。Model 代表的是应用程序的商业逻辑（通过 JavaBean、EJB 等组件实现），View 是系统的显示接口（使用 JSP 来输出 HTML），Controller 是提供应用程序的处理过程控制（通常是 Servlet），通过这种设计模型把应用逻辑、处理程序和显示接口分成不同的组件实现，这些组件可以进行交互和重用来弥补 Model 1 的不足(见图 18-4)。

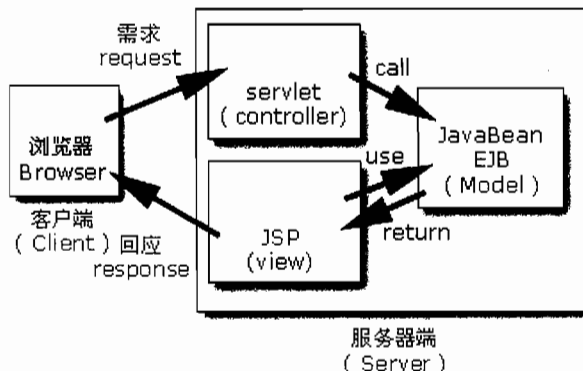


图 18-4 Model 2 MVC 架构

MVC 并不是新概念，早在 smalltalk-80 时代就已经使用这个概念来设计应用程序，因为网站应用系统普及化，MVC 具有提升组件重复使用性、事件及动作由中心控制及维护修改容易，尤其构建大型系统的时候，更需要有一个完善的框架(Framework)来开发。

这种做法的优点为：

- 开发流程更为明确：使用 Model 2 的设计模式可以完全切开显示端与商业逻辑端的开发，让美工设计人员与程序员可以专注于本身的工作，有利于大型系统的开发。
- 核心的程序管控：由 Controller 控制整个流程，可以减少 JSP 需要撰写许多条件判断逻辑及流程管控等等的程序代码。
- 维护容易：不论是后端商业逻辑对象或前端的网页呈现，都通过控制中心来掌控，如果有商业逻辑的变更，可以轻易地修改 Model 端的程序，而不用去修改相关的 JSP 文件。

这种设计模式的缺点是：

- 学习时间较长：各家公司都有本身的 MVC 架构，工程师都需要花更多的时间去熟悉了解他们的流程与概念。
- 开发时间较长：因为需要设计 MVC 各对象彼此的数据交换格式与方法，会需要更多的时间在系统设计之上。

MVC 本身就是一个非常复杂的系统，所以采用 MVC 实现交互式的网站时，最好选一个现成的 MVC 框架，在此之下进行开发，从而取得事半功倍的效果，也可以直接寻找具有相关经验的工程师来开发，进而减少学习时间的影

响。目前有很多可供使用的 MVC 框架，由于 Struts 具有完整的文件，并且公开源代码，因此是

JSP2.0 技术手册

目前全世界最多 Java 族群使用的框架。笔者无法在短短一两个章节里详细地介绍 Struts 的使用，而且目前市面上已经有许多专门讨论 Struts 的书籍，因此本书将不讨论 Struts 的内容。

接下来举个范例程序，分别使用 Model 1 和 Model 2 的模块，相信读者很快就能了解它们之间的差异性。

18-3 Model 1 和 Model 2 的范例程序

本节笔者将举几个范例程序，希望各位读者能够通过范例程序中的写法来了解如何套用 Model 1 和 Model 2，并且制作一个完全符合 MVC (Model-View-Controller) 架构的购物车程序。不过笔者会将重心放在介绍 Model 2 的写法，因为 Model 1 的写法和一般 JSP+JavaBean 的写法相同，在此不多加说明。

18-3-1 Model 2 的范例程序——Hello World

首先举个 Model 2 的 Hello World 程序。请求实际上从 Servlet (*Model2Hello.class*) 进入，然后再转交给 JSP (*Model2Hello.jsp*):

■ *Model2Hello.java*

```
package tw.com.javaworld.CH18;

import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Model2Hello extends HttpServlet {

    public void init(ServletConfig config) throws ServletException {
        super.init(config);
    }

    public void service(HttpServletRequest request,
                        HttpServletResponse response)
        throws ServletException, IOException {

        response.setContentType("text/html");
        String Message = "Hello World";
        String jsp = "/CH18/Model2Hello.jsp";
        request.setAttribute("message", Message);

        RequestDispatcher rd =
            getServletContext().getRequestDispatcher(jsp);
        rd.forward(request, response);
    }

    public void destroy() {
```

JSP2.0 技术手册

```
}  
}
```

当 *Model2Hello* 被执行时，它会产生一个 Hello World 的字符串，存入到 Request 范围的 Message 变量。使用 RequestDispatcher 接口的 forward() 方法，将请求转交给 *Model2Hello.jsp*。而 RequestDispatcher 接口在本节当中是一个关键的角色，它主要扮演流程控制的分派员(Dispatcher)，我们稍后会详细介绍它。

不知道读者有没有发现，在 *Model2Hello* 的程序中并没有任何的输出，如 out.println("XXX")。因为我们将输出的工作交由 JSP 网页负责，而 Servlet 用来产生数据和流程控制。

■ Model2Hello.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>  
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>  
  
<html>  
<head>  
  <title>CH18 - Model2Hello.jsp</title>  
</head>  
<body>  
  
  <h2>Model 2 范例 - Hello World</h2>  
  
  The Message is : <font color="red">${requestScope.message}</font>  
  
</body>  
</html>
```

Model2Hello.jsp 只用来显示数据的结果。这里我们利用到 getAttribute() 方法，这方法有点像 getParameter()，不过所代表的意义不同，详细解说请参考“第五章：隐含对象 (Implicit Object)”。程序执行结果如图 18-5。



图 18-5 Model2Hello Servlet 的执行结果

18-3-2 RequestDispatcher 接口

之前已经谈到 Model 2 都是利用 Servlet 来做流程控制(Flow Control)，本小节就说明如何使用 Servlet 做流程控制的工作。Servlet API 2.1 有一个 RequestDispatcher 接口，它能够允许你将请求转交给另一个 JSP 网页、Servlet 或者将数据(如：JSP 网页、Servlet、HTML 等等)的输出一并加入到原来的输出流中。RequestDispatcher 对象有下列两种方法可以取得：

```
RequestDispatcher rd = request.getRequestDispatcher("Checkout.jsp");
```

或者

```
RequestDispatcher rd = getServletContext().getRequestDispatcher (
    "/Store/Checkout.jsp")
```

RequestDispatcher 接口提供两种方法：include()和 forward()：

```
public void include(HttpServletRequest , HttpServletResponse)
public void forward(HttpServletRequest , HttpServletResponse)
```

你可以使用 forward()方法将目前的请求服务转交到另一个 JSP 网页或是 Servlet；或者也可以通过 include()的方法，将它的内容一并包含到原来的 Servlet 中。

注意

虽然我们将请求服务转向到另一个 JSP 网页，但是浏览器的 URL 不会有任何的变化，那是因为转向或包含的运作皆在服务器中处理，因此浏览器的 URL 不会有任何的变化。如图 18-5 所示，我已经将请求转向到 Mode2Hello.jsp，不过浏览器上的 URL 维持原来 Servlet 的 URL。

18-3-3 Model 2 范例程序——Shoppingcart

接下来将举一个较为复杂的范例程序——Shoppingcart。这个范例当中，我们写了三个 JSP 网页：EShop.jsp、Cart.jsp、Checkout.jsp；一个 JavaBean：Book.java 和一个 Servlet：ShoppingServlet.java。如表 18-1 所示：

表 18-1

名 称	功能说明
EShop.jsp	显示商店所提供的商品目录
Cart.jsp	显示购物车的内容
Checkout.jsp	显示用户欲购买的物品清单和价格
Book.java	内容为书籍的属性：书名、作者、出版商、价格、数量。它是一个 JavaBean
ShoppingServlet.java	它除了新增、移除购物车内的物品和结账的功能之外，并且也扮演流程控制的角色。它本身是 Servlet

■ EShop.jsp

```
<%@ page contentType="text/html; charset=GB2312" %>
```

```

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<html>
<head>
  <title>CH18 - EShop.jsp</title>
</head>
<body>

<h2>Model 2 范例 - Shoppingcart</h2>
<font size="+3">网络书店</font>
<hr><p>

<table border="1" width="631">
  <tr bgcolor="#999999">
    <td width="174"><div align="center"><b>书名</b></div></td>
    <td width="101"><div align="center"><b>作者</b></div></td>
    <td width="57"><div align="center"><b>出版社</b></div></td>
    <td width="93"><div align="center"><b>价格</b></div></td>
    <td width="47"><div align="center"><b>数量</b></div></td>
    <td width="119"><div align="center"><b></b></div></td>
  </tr>
  <form name="shoppingForm" action="ShoppingServlet" method="POST">
    <tr>
      <td width="174"><div align="center">JSP 2.0 技术手册</div></td>
      <td width="101"><div align="center">林上杰</div></td>
      <td width="57"><div align="center">电子工业出版社</div></td>
      <td width="47"><div align="right">50</div></td>
      <td width="93">数量:
        <input type="text" name="quantity" size="3" value=1>
      </td>
      <td width="119">
        <input type="submit" name="Submit" value="放入购物车">
      </td>
    </tr>
    <input type="hidden" name="name" value="JSP 2.0 技术手册">
    <input type="hidden" name="author" value="林上杰">
    <input type="hidden" name="publisher" value="电子工业出版社">
    <input type="hidden" name="price" value="50">
    <input type="hidden" name="action" value="ADD">
  </form>
  <form name="shoppingForm" action="ShoppingServlet" method="POST">
    <tr>
      <td width="174"><div align="center">紫牛——让产品自己说故事</div></td>
      <td width="101"><div align="center">赛斯·高汀</div></td>
      <td width="57"><div align="center">商智</div></td>
      <td width="47"><div align="right">237</div></td>
      <td width="93">数量:
        <input type="text" name="quantity" size="3" value=1>
      </td>
      <td width="119">

```

JSP2.0 技术手册


```

        <input type="submit" name="Submit" value="放入购物车">
    </td>
</tr>
<input type="hidden" name="name" value="紫牛——让产品自己说故事">
<input type="hidden" name="author" value="赛斯·高汀">
<input type="hidden" name="publisher" value="商智">
<input type="hidden" name="price" value="237">
<input type="hidden" name="action" value="ADD">
</form>
<form name="shoppingForm" action="ShoppingServlet" method="POST">
    <tr>
        <td width="174"><div align="center">1421-中国发现世界</div></td>
        <td width="101"><div align="center">孟西士</div></td>
        <td width="57"><div align="center">远流</div></td>
        <td width="47"><div align="right">470</div></td>
        <td width="93">数量:
            <input type="text" name="quantity" size="3" value="1">
        </td>
        <td width="119">
            <input type="submit" name="Submit" value="放入购物车">
        </td>
    </tr>
    <input type="hidden" name="name" value="1421-中国发现世界">
    <input type="hidden" name="author" value="孟西士">
    <input type="hidden" name="publisher" value="远流">
    <input type="hidden" name="price" value="470">
    <input type="hidden" name="action" value="ADD">
</form>
<form name="shoppingForm" action="ShoppingServlet" method="POST">
    <tr>
        <td width="174"><div align="center">如何打败可口可乐</div></td>
        <td width="101"><div align="center">翟若适</div></td>
        <td width="57"><div align="center">联合文学</div></td>
        <td width="47"><div align="right">180</div></td>
        <td width="93">数量:
            <input type="text" name="quantity" size="3" value="1">
        </td>
        <td width="119">
            <input type="submit" name="Submit" value="放入购物车">
        </td>
    </tr>
    <input type="hidden" name="name" value="如何打败可口可乐">
    <input type="hidden" name="author" value="翟若适">
    <input type="hidden" name="publisher" value="联合文学">
    <input type="hidden" name="price" value="180">
    <input type="hidden" name="action" value="ADD">
</form>
<form name="shoppingForm" action="ShoppingServlet" method="POST">
    <tr>
        <td width="174"><div align="center">菲奥利娜逆势出击</div></td>

```



```

<td width="101"><div align="center">彼得·鲍洛斯</div></td>
<td width="57"><div align="center">商周出版</div></td>
<td width="47"><div align="right">300</div></td>
<td width="93">数量:
  <input type="text" name="quantity" size="3" value="1">
</td>
<td width="119">
  <input type="submit" name="Submit" value="放入购物车">
</td>
</tr>
<input type="hidden" name="name" value="菲奥利娜逆势出击">
<input type="hidden" name="author" value="彼得·鲍洛斯">
<input type="hidden" name="publisher" value="商周出版">
<input type="hidden" name="price" value="300">
<input type="hidden" name="action" value="ADD">
</form>
</table>
<p>
  <jsp:include page="Cart.jsp" flush="true" />
</body>
</html>

```



图 18-6 EShop.jsp 的执行结果

EShop.jsp 列出了所有商品的数据, 如图 18-6 所示。假若用户要购买商品时, 按下【放入购物车】的按钮时, 它会将数据传送到 *ShoppingServlet.java*, 将选定的商品加入至购物车中。*EShop.jsp* 也将 *Cart.jsp* 包含进来, *Cart.jsp* 是显示购物车内容的 JSP 网页。接下来看 *Cart.jsp*。

■ *Cart.jsp*

```
<%@ page import="java.util.*,tw.com.javaworld.CH18.Book" %>
```

```
<%@ page contentType="text/html; charset=GB2312" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<html>
<head>
  <title>CH18 - Cart.jsp</title>
</head>
<body>

<c:set var="buylist" value="${sessionScope.shoppingcart}" />
<c:if test="${!empty buylist}" >

<br>
<h2>目前您购物车的内容如下: </h2><br>
<table border="1" width="631">
  <tr bgcolor="#999999">
    <td width="194"><div align="center"><b>书名</b></div></td>
    <td width="81"><div align="center"><b>作者</b></div></td>
    <td width="93"><div align="center"><b>出版社</b></div></td>
    <td width="57"><div align="center"><b>价格</b></div></td>
    <td width="47"><div align="center"><b>数量</b></div></td>
    <td width="119"><div align="center"><b></b></div></td>
  </tr>

  <c:forEach items="${buylist}" var="order" varStatus="status" >

<tr>
  <td><b>${order.name}</b></td>
  <td><b>${order.author}</b></td>
  <td><b>${order.publisher}</b></td>
  <td><b><div align="right">${order.price}</div></b></td>
  <td><b><div align="right">${order.quantity}</div></b></td>
  <td>
    <form name="deleteForm" action="ShoppingServlet" method="POST">
      <input type="submit" value="Delete">
      <input type="hidden" name="del" value='${status.index}'>
      <input type="hidden" name="action" value="DELETE">
    </form>
  </td>
</tr>

</c:forEach>

</table>
<p>
  <form name="checkoutForm" action="ShoppingServlet" method="POST">
    <input type="hidden" name="action" value="CHECKOUT">
    <input type="submit" name="Checkout" value="付款结账">
  </form>
</c:if>
```

Cart.jsp 专门负责将购物车中的商品内容显示出来, 下列是 *Cart.jsp* 最主要的程序部分。首先使用 `<c:set>` 将 Session 范围中的 `shoppingcart` 对象存放在 `buylist` 的变量中, 其中

shoppingcart 对象的类型为 Vector。

然后再使用<c:forEach>将 buylist 中的 Book 类一一取出来，并且存放在 order 变量中，所以 order 的类型为 Book。每一个 Book 类就是代表一本书，而每一本书又有很多属性，如：书名、作者、出版社等等。最后直接使用 \${order.name}、\${order.author}、\${order.publisher}、\${order.price}和\${order.quantity}，分别取得每一本书的书名、作者、出版社、单价、数量等等的属性。

主要的片段程序代码，如下所示：

```
<c:set var="buylist" value="${sessionScope.shoppingcart}" />
<c:forEach items="${buylist}" var="order" varStatus="status" >

<tr>
<td><b>${order.name}</b></td>
<td><b>${order.author}</b></td>
<td><b>${order.publisher}</b></td>
<td><b><div align="right">${order.price}</div></b></td>
<td><b><div align="right">${order.quantity}</div></b></td>
<td>
<form name="deleteForm" action="ShoppingServlet" method="POST">
<input type="submit" value="Delete">
<input type="hidden" name="del" value='${status.index}'>
<input type="hidden" name="action" value="DELETE">
</form>
</td>
</tr>
</c:forEach>
```

Cart.jsp 的执行结果如图 18-7。

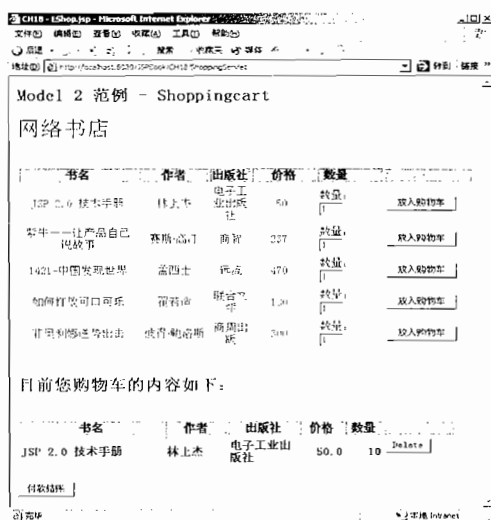


图 18-7 新增商品至购物车中的执行画面

下面来看 *ShoppingServlet.java*。

■ *ShoppingServlet.java*

```
package tw.com.javaworld.CH18;

import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ShoppingServlet extends HttpServlet {
    public void init(ServletConfig conf) throws ServletException {
        super.init(conf);
    }

    public void doPost(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
        HttpSession session = req.getSession(false);

        // 若 session 为 null 时, 程序将导向 Error.html
        if (session == null) {
            res.sendRedirect("Error.html");
        }
        Vector buylist = (Vector) session.getAttribute("shoppingcart");
        String action = req.getParameter("action");

        if (!action.equals("CHECKOUT")) {

            // 删除购物车中的书籍
            if (action.equals("DELETE")) {
                String del = req.getParameter("del");
                int d = (new Integer(del)).intValue();
                buylist.removeElementAt(d);
            }

            // 新增书籍至购物车中
            else if (action.equals("ADD")) {
                boolean match = false;

                // 取得后来新增的书籍
                Book abook = getBook(req);

                // 新增第一本书籍至购物车时
                if (buylist == null) {
                    buylist = new Vector();
                    buylist.addElement(abook);
                } else {
                    for (int i = 0; i < buylist.size(); i++) {
                        Book book = (Book) buylist.elementAt(i);
```

JSP2.0 技术手册

```

        // 假若新增的书籍和购物车的书籍一样时
        if (book.getName().equals(abook.getName())) {
            book.setQuantity(
                book.getQuantity() + abook.getQuantity());
            buylist.setElementAt(book, i);
            match = true;
        } //end of if name matches
    } // end of for

    // 假若新增的书籍和购物车的书籍不一样时
    if (!match)
        buylist.addElement(abook);
    }

    session.setAttribute("shoppingcart", buylist);
    String url = "/CH18/EShop.jsp";
    ServletContext sc = getServletContext();
    RequestDispatcher rd = sc.getRequestDispatcher(url);
    rd.forward(req, res);
}

// 结账, 计算购物车书籍价钱总数
else if (action.equals("CHECKOUT")) {
    float total = 0;
    for (int i = 0; i < buylist.size(); i++) {
        Book order = (Book) buylist.elementAt(i);
        float price = order.getPrice();
        int quantity = order.getQuantity();
        total += (price * quantity);
    }

    String amount = new Float(total).toString();
    req.setAttribute("amount", amount);
    String url = "/CH18/Checkout.jsp";
    ServletContext sc = getServletContext();
    RequestDispatcher rd = sc.getRequestDispatcher(url);
    rd.forward(req, res);
}

private Book getBook(HttpServletRequest req) {
    String name = encoding(req.getParameter("name"));
    String quantity = encoding(req.getParameter("quantity"));
    String author = encoding(req.getParameter("author"));
    String publisher = encoding(req.getParameter("publisher"));
    String price = encoding(req.getParameter("price"));
}

```

```

        Book bk = new Book();

        bk.setName(name);
        bk.setAuthor(author);
        bk.setPublisher(publisher);
        bk.setPrice((new Float(price)).floatValue());
        bk.setQuantity((new Integer(quantity)).intValue());
        return bk;
    }

    private String encoding(String str) {
        try {
            str = new String(str.getBytes("ISO-8859-1"), "GB2312");
        } catch (UnsupportedEncodingException uee) {
            System.out.println("UnsupportedEncodingException: " +
                               uee.getMessage());
        }

        return str;
    }
}

```

ShoppingServlet.java 的程序代码比较长, 不过它主要就是做新增、删除和结账三个大功能(其实你也可以自行加入修改的功能)。*ShoppingServlet.java* 程序中, 要注意的是它流程的控制, 当在新增或删除书籍的动作执行完后, 将购物车的 *Vector* 对象加入至 *session* 对象中, 并且命名为 *shoppingcart*:

```
session.setAttribute("shoppingcart", buylist);
```

然后将请求转向至 *EShop.jsp*:

```

String url="/CH18/EShop.jsp";
ServletContext sc = getServletContext();
RequestDispatcher rd = sc.getRequestDispatcher(url);
rd.forward(req, res);

```

同样地, 若是执行完成结账后, 就将请求转向到 *Checkout.jsp*:

```

String url="/CH18/Checkout.jsp";
ServletContext sc = getServletContext();
RequestDispatcher rd = sc.getRequestDispatcher(url);
rd.forward(req,res);

```

■ *Book.java*

```

package tw.com.javaworld.CH18;

import java.io.Serializable;

public class Book implements Serializable {

```



```
private String name;
private String author;
private String publisher;
private float price;
private int quantity;

public Book() {
}

public String getName() {
    return name;
}
public String getAuthor() {
    return author;
}
public String getPublisher() {
    return publisher;
}
public void setPrice(float newPrice) {
    price = newPrice;
}
public float getPrice() {
    return price;
}
public void setQuantity(int newQuantity) {
    quantity = newQuantity;
}
public int getQuantity() {
    return quantity;
}
public void setPublisher(String newPublisher) {
    publisher = newPublisher;
}
public void setAuthor(String newAuthor) {
    author = newAuthor;
}
public void setName(String newName) {
    name = newName;
}
}
```

Book.java 中存放书名(name)、作者(author)、出版社(publisher)、价格(price)、数量(quantity)等属性,我们利用 JavaBean 的 `getProperty` 和 `setProperty` 来设定或存取书籍的属性值。将来若想新增书籍的页数、分类等属性,只需要在 *Book.java* 中新增 `page`、`catalog` 属性就行了,这就是对象的好处之一,让程序功能的弹性增加许多。最后就是结账时,所列出购买商品的清单——*Checkout.jsp*。

■ *Checkout.jsp*

```
<%@ page import="java.util.*,tw.com.javaworld.CH18.Book" %>
<%@ page contentType="text/html;charset=GB2312" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

```
<html>
<head>
  <title>CH18 - Checkout.jsp</title>
</head>
<body>

<br>

<center>
  <h2>网络书店 - 结账</h2><br>
  <table border="1" width="631">
    <tr bgcolor="#999999">
      <td width="194"><div align="center"><b>书名</b></div></td>
      <td width="81"><div align="center"><b>作者</b></div></td>
      <td width="57"><div align="center"><b>出版社</b></div></td>
      <td width="93"><div align="center"><b>价格</b></div></td>
      <td width="47"><div align="center"><b>数量</b></div></td>
      <td width="119"><div align="center"><b></b></div></td>
    </tr>

    <c:set var="buylist" value="${sessionScope.shoppingcart}" />
    <c:set var="amount" value="${requestScope.amount}" />

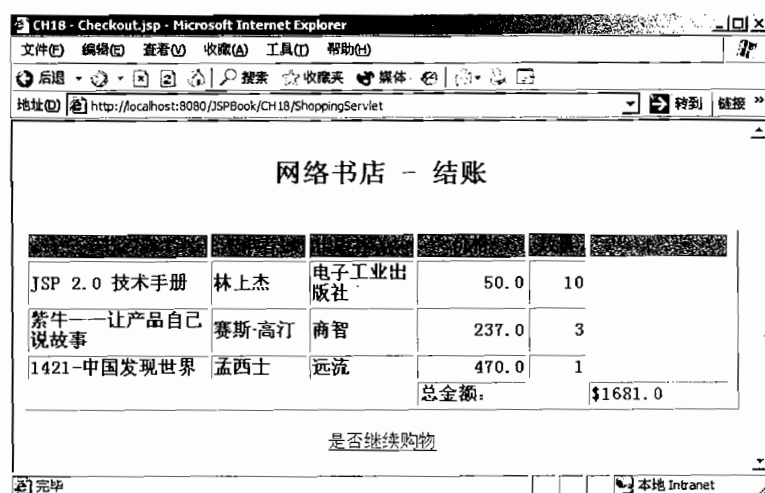
    <c:forEach items="${buylist}" var="order" varStatus="status" >

      <tr>
        <td><b>${order.name}</b></td>
        <td><b>${order.author}</b></td>
        <td><b>${order.publisher}</b></td>
        <td><b><div align="right">${order.price}</div></b></td>
        <td><b><div align="right">${order.quantity}</div></b></td>
      </tr>

    </c:forEach>

    <tr>
      <td></td>
      <td></td>
      <td></td>
      <td><font color="red"><b>总金额: </b></font></td>
      <td></td>
      <td><font color="red"><b>\${requestScope.amount}</b></font></td>
      <td></td>
    </tr>
  </table>
  <p>
    <a href="../CH18/EShop.jsp">是否继续购物</a>
  </p>
</center>
</body>
</html>
```

`Checkout.jsp` 列出用户决定购买书籍的清单, `Checkout.jsp` 的执行结果如图 18-8 所示。



书名	作者	出版社	单价	数量
JSP 2.0 技术手册	林上杰	电子工业出版社	50.0	10
紫牛——让产品自己说故事	赛斯·高汀	商智	237.0	3
1421-中国发现世界	孟西士	远流	470.0	1
总金额:			\$1681.0	

[是否继续购物](#)

图 18-8 `Checkout.jsp` 的执行结果

A

附录 A

安装 Linux 执行环境

本附录将以 Step by Step 的方式，说明在 Linux 下，如何安装 Servlet/JSP 执行环境和使用本书之范例程序。本附录将分以下部分来分别介绍：

- A-1 安装 J2SDK 1.4.2
- A-2 安装 Tomcat 5.0.16
- A-3 安装 JSPBook 站台范例
- A-4 安装 Ant 1.6
- A-5 安装 Apache 2.0.48 + Tomcat 5.0.16

JSP2.0 技术手册

A-1 安装 J2SDK 1.4.2

只要我们需要执行、开发任何有关 Java 程序时，首先都须先安装 Java 2 Software Development Kit，简称 J2SDK 或者又称 JDK(Java Development Kit)，它主要包含：

- Java API
- Java Compiler “javac”
- Java Debugger
- Java Plug-in
- Java HotSpot Client Virtual Machine
- Java 2 RE(Java 2 Runtime Environment)

■ Java Compiler 和 Java Debugger

Java Compiler 和 Java Debugger 主要用来开发 Java 程序，因此 Java 2 SDK 也可以算是最简易的 Java 程序开发工具。

■ Java Plug-in

Java Plug-in 主要让浏览器如 Internet Explorer 或是 Netscape 用来执行 Applet 的软件。

■ Java HotSpot Client Virtual Machine

Java HotSpot Client Virtual Machine 就是俗称的 JVM，据 JavaSoft 官方网站的数据宣称，Java 2 SDK 1.3 加入 HotSpot 是为了增进 JVM 的执行率效，他们甚至还说：某些情况下 Java 的编译、执行时间会和 C++ 并驾齐驱，并且它的性能比之前 1.0.1 版要快上 30%。但是，老实说 Java 最另人诟病的问题还是在于 Java 的执行效率太差，希望未来在性能的表现上能有更突出的表现。

■ Java 2 RE(Java 2 Runtime Environment)

简单来说，JRE(Java Runtime Environment)就是一个标准 Java 应用程序的执行环境。JavaSoft 官方网站中，也有单独将这一部分让用户方便下载。原因在于，假若今天用户使用一套 Java 所写的应用软件或是需要执行 Applet 时，只要先安装 JRE 即可，无须再去下载一个庞大的 Java 2 SDK 来安装使用，除非你本身也想要从事 Java 程序的开发，才需要一套集成开发和执行环境的软件。

Tomcat 5.0.16 运作时，必须要有 J2SDK 1.3.1 或以上的版本，而 J2SDK 可以从 <http://www.javasoft.com> 的网站自行免费下载，其中 J2SDK 提供有 Solaris、Linux、Windows 三种平台的版本。目前 Linux 平台的最新 J2SDK 版本为 J2SDK 1.4.2，如图 A-1：

JSP2.0 技术手册

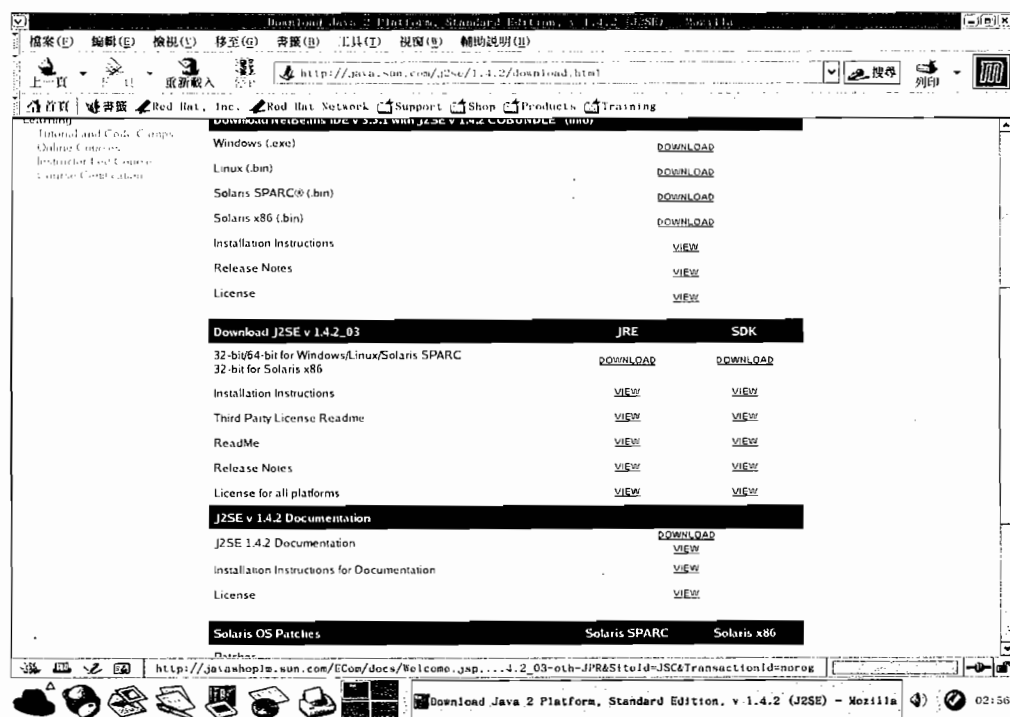


图 A-1 下载 J2SDK 1.4.2

或直接使用本书 CD 光盘中的 J2SDK 1.4.2_03, 软件名称为: `j2sdk-1_4_2_03-linux-i586.bin`。

第一步: 安装 `j2sdk-1_4_2_03-linux-i586.bin`;

将 `j2sdk-1_4_2_03-linux-i586.bin` 放至 `/usr/local/` 目录下, 依照下列步骤来安装:

- `cd /usr/local/`
- `chmod 755 j2sdk-1_4_2_03-linux-i586.bin`
- `./j2sdk-1_4_2_03-linux-i586.bin`

打完以上指令后会看到版权声明, 然后输入 `yes` 即可。

第二步: 设定 J2SDK 1.4.2_03;

先到自己的目录底下根据自己的 shell 在登录文件中设定, 例如: 使用 `bash` 的用户, 则用编辑软件编辑 `.bash_profile`, 如下:

- `vi ~/.bash_profile`

在 `.bash_profile` 的最后面加上:

- `export JAVA_HOME=/usr/local/j2sdk1.4.2_03`
- `export PATH=$PATH:$JAVA_HOME/bin`
- `export CLASSPATH=.:$JAVA_HOME/lib/tools.jar`

注意

1. CLASSPATH 的设定中, 冒号 “:” 用来分开两路径, 切勿任意空格;
2. CLASSPATH 的设定中有一个点 “.”。

设定完后在 console 下键入:

- source ~/.bash_profile
- java -version

可以看到如图 A-2:

```
root@localhost koji1# java -version
java version "1.4.2_03"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.2_03-b02)
Java HotSpot(TM) Client VM (build 1.4.2_03-b02, mixed mode)
root@localhost koji1# _
```

图 A-2 java -version 之结果

第三步: 测试 J2SDK。

撰写一个 *HelloWorld.java* 程序, 放置在 *~/HelloWorld.java* 中。

■ HelloWorld.java

```
public class HelloWorld {

    public static void main(String[] args) {
        System.out.println("Hello World");
    }

}
```

然后打开命令提示符, 在 *~/* 下输入 *javac HelloWorld.java*, 然后再输入 *java HelloWorld*, 执行 *HelloWorld* 程序, 假若顺利成功, 则会显示 “Hello World”, 如图 A-3 所示:

```
[koji@localhost koji1]$ javac HelloWorld.java
[koji@localhost koji1]$ java HelloWorld
Hello World
[koji@localhost koji1]$ _
```

图 A-3 编译且执行 HelloWorld 程序

成功安装 J2SDK 1.4.2_03 之后, 紧接下来安装 Tomcat 5.0.16。

A-2 安装 Tomcat 5.0.16

Tomcat 目前版本为 5.0.16, 它是由 JavaSoft 和 Apache 开发团队共同提出合作计划 (Apache Jakarta Project) 下的产品。Tomcat 能支持 Servlet 2.4 和 JSP 2.0 并且是免费使用。

Tomcat 5.0.16 可以从 <http://jakarta.apache.org/tomcat/index.html> 网站自行免费下载, 或者可以直接使用本书 CD 光盘中的 Tomcat 5.0.16, 软件名称为: *jakarta-tomcat-5.0.16.tar.gz.tar*。

第一步: 安装 *jakarta-tomcat-5.0.16.tar.gz.tar*;

下载完 *jakarta-tomcat-5.0.16.tar.gz.tar* 后, 笔者打算将 Tomcat 安装在 */usr/local* 下, 因

此, 在 `/usr/local` 下建立新的文件夹 `Jakarta`, 再将 `jakarta-tomcat-5.0.16.tar.gz` 解压缩至 `/usr/local/Jakarta` 目录下, 指令如下:

- `mkdir /usr/local/Jakarta`
- `tar zxvf jakarta-tomcat-5.0.16.tar.gz -C /usr/local/Jakarta`

第二步: 执行 Tomcat;

在 Linux 下执行 Tomcat 非常简单。只要执行 `{Tomcat_Install}/bin` 中的 `catalina.sh` 即可(例如: `/usr/local/jakarta/jakarta-tomcat-5.0.16/bin/catalina.sh`), 指令如下:

- `cd /usr/local/jakarta/jakarta-tomcat-5.0.16/bin/`
- `./catalina.sh start`

注意

执行前, 请确定已设定 `JAVA_HOME=/usr/local/j2sdk1.4.2_03`, 因为启动 Tomcat 时必须用到。

第三步: 测试 Tomcat。

打开浏览器如: Netscape, 输入 `http://主机地址:8080`, 假若 Tomcat 安装成功, 则会看到如图 A-4 的情形:

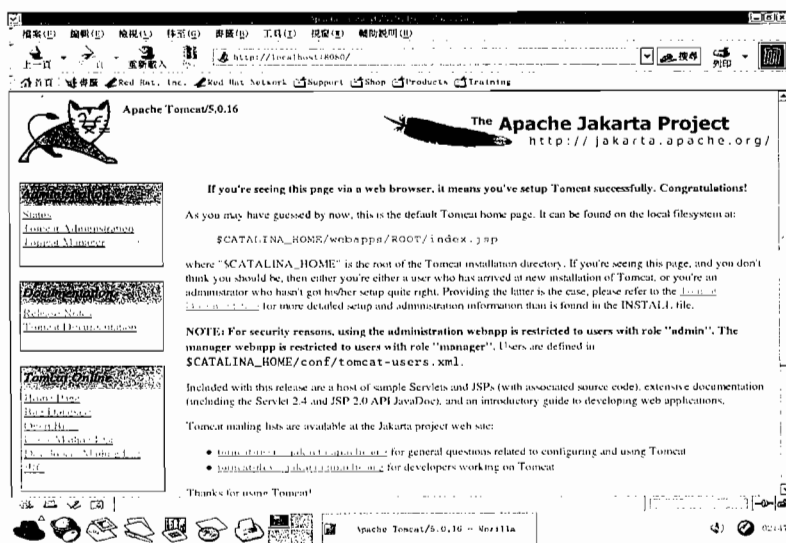


图 A-4 连接 `http://主机地址:8080/`, 测试 Tomcat 5.0.16

本书“第十二章: JSP 执行环境与开发工具”对于 Tomcat 的使用及设定有更详细的介绍。

A-3 安装 JSPBook 站台范例

读者可以在 CD 光盘中找到本书的范例, 程序文件名为 `JSPBook.war`。

JSP2.0 技术手册

第一步：安装 *JSPBook.war*；

安装的方法很简单，只要将 *JSPBook.war* 移至 *{Tomcat_Install}/webapps/* 目录下(例如：*/usr/local/jakarta/jakarta-tomcat-5.0.16/webapps/JSPBook.war*)，然后 *JSPBook.war* 会自动被 Tomcat 解压缩成 JSPBook 的目录，如图 A-5：



图 A-5 安装 JSPBook.war

第二步：设定 JSPBook 站台；

在 Tomcat 上建立一个 JSPBook 站台时，我们须要修改 Tomcat 的 *server.xml* 文件，*server.xml* 位于 *{Tomcat_Install}/conf/server.xml* (例如：*/usr/local/jakarta/jakarta-tomcat-5.0.16/conf/server.xml*)。

■ *server.xml*

```

<!-- Tomcat Root Context -->
<!--
  <Context path="/" docBase="ROOT" debug="0">
  -->
  <Context path="/JSPBook" docBase="JSPBook" debug="0"
    crosscontext="true" reloadable="true" >
  </Context>
</Host>
</Engine>
</Service>
</Server>

```

这部分主要是设定 JSPBook 站台，其中 *path="/JSPBook"* 代表网域名称，即 *http://IP_DomainName/JSPBook*；*docBase="JSPBook"* 代表站台的目录位置，即 *{Tomcat_Install}/webapps/JSPBook*；*debug* 则是设定 debug level，0 表示最少的信息，9 表示提供最多的信息；*reloadable* 则表示 Tomcat 在执行时，修改过 *class*、*web.xml* 时，都会自动重新加载，不需要重新启动 Tomcat。

第三步：执行 JSPBook 站台（见图 A-6）；

第四步：JSPBook 站台目录结构。

JSPBook 目录下包含：

- (1) 各章节的 HTML/JSP 程序；
- (2) *dist* 目录：存放 JSPBook 站台压缩后的 *JSPBook.war*；
- (3) *build.xml*：Ant 文件；
- (4) *WEB-INF* 目录：包含 *classes*、*lib*、*tags* 和 *src*；
- (5) *src* 目录：存放范例的源程序，如：JavaBean、Filter、Servlet 等等。

JSP2.0 技术手册

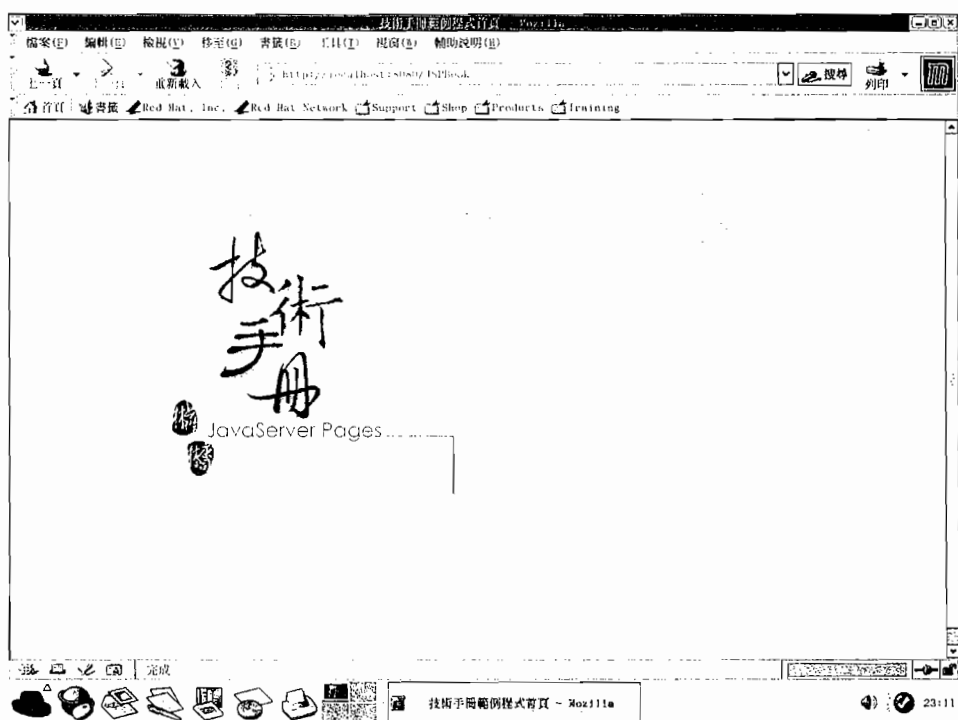


图 A-6 执行 JSPBook 站台

图 A-7 为 JSPBook 站台目录结构：

```

C:\root\localhost\JSPBook10 - ls
build.xml          error.html         login.html
                    index.html         meta.html
C:\root\localhost\JSPBook10 _
  
```

图 A-7 JSPBook 站台目录结构

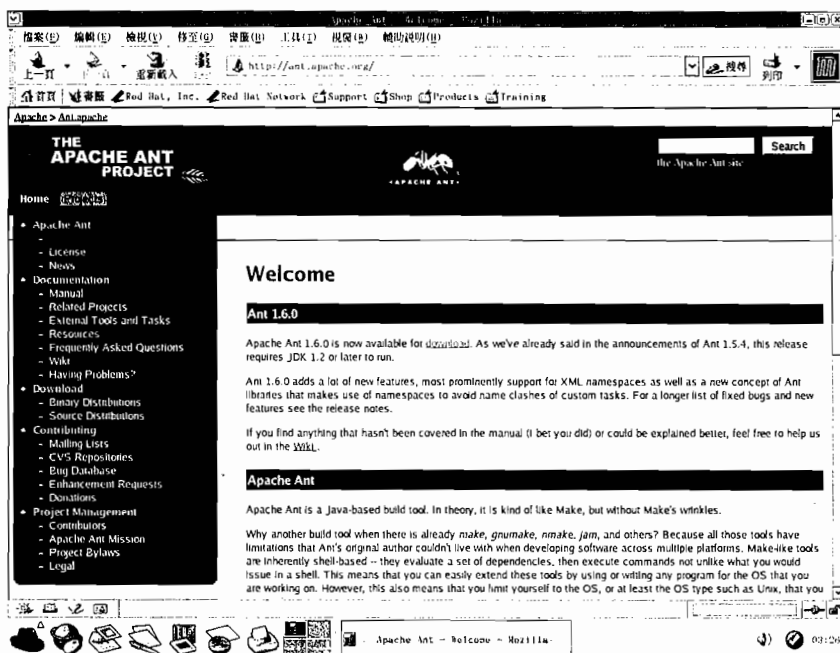
A-4 安装 Ant 1.6

修改 JSP 程序时，Tomcat 会自动将 JSP 重新转译为 Servlet，并且编译 Servlet。但是，假若修改 Servlet、JavaBean 或 Filter 时，我们就必须自行先编译它们，然后再将它们重新部署至 `WEB-INF\classes` 下。

为了方便编译这些程序，笔者提供 JSPBook 站台的 `build.xml` 文件，因此，建议读者先安装 Ant 1.6，并且学习使用 Ant。

目前 Ant 最新版本为 1.6，读者可以自行至 <http://ant.apache.org> 下载最新版本，如图 A-8，或者直接使用本书 CD 光盘中的 Ant 1.6，软件名称为：`apache-ant-1.6.0-bin.tar.gz`。

JSP2.0 技术手册

图 A-8 <http://ant.apache.org>

第一步：将 *apache-ant-1.6.0-bin.tar.gz* 解压缩：

下载完 *apache-ant-1.6.0-bin.tar.gz* 后，笔者打算将 Tomcat 安装在 */usr/local* 下，因此在 */usr/local* 下建立新的文件夹 *ant*，再将 *apache-ant-1.6.0-bin.tar.gz* 解压缩至 */usr/local/ant* 目录下，指令如下：

- `mkdir /usr/local/ant`
- `tar zxvf apache-ant-1.6.0-bin.tar.gz -C /usr/local/ant`

第二步：设定 Ant 1.6；

先到自己的目录底下根据自己的 shell 在登录文件中设定，例如：使用 bash 的用户，则用编辑软件编辑 *.bash_profile*，如下：

- `vi ~/.bash_profile`

在 *.bash_profile* 的最后面加上：

- `export ANT_HOME=/usr/local/ant/apache-ant-1.6.0`
- `export PATH=$PATH:$ANT_HOME/bin`

设定完后在 console 下键入：

- `source ~/.bash_profile`

第三步：测试 Ant 1.6；

打开命令提示符，输入 `ant -version`，假若执行成功，会有如图 A-9 的结果：


```
lkoji@localhost koji1$ ant -version
Apache Ant version 1.6.0 compiled on December 18 2003
lkoji@localhost koji1$ _
```

图 A-9 测试 Ant 1.6

第四步：使用 Ant 1.6 编译 *JSPBook/WEB-INF/src* 中的程序。

要编译修改过的 *JSPBook/WEB-INF/src* 中的程序时，首先打开命令提示符，然后移至 JSPBook 站台的所在目录，例如：*/usr/local/jakarta/jakarta-tomcat-5.0.16/webapps/JSPBook*。

然后执行 ant，它会先自动找到 *JSPBook/build.xml* 文件，然后根据 *build.xml* 的设定，编译 */usr/local/jakarta/jakarta-tomcat-5.0.16/webapps/JSPBook/WEB-INF/src* 目录下的所有 Java 源文件，然后产生的类文件夹存放至 */usr/local/jakarta/jakarta-tomcat-5.0.16/webapps/JSPBook/WEB-INF/classes* 目录下。图 A-10 为执行 ant 指令之结果：

```
lroot@localhost JSPBook1# ant
Build file: build.xml

init:
  echo! Init Complete !
  echo! ant home = /usr/local/ant/apache-ant-1.6.0
  echo! java home = /usr/java/j2sdk1.4.2_03/jre
  echo! user home = /root

compile:
  echo! Compilation Complete !

BUILD SUCCESSFUL
Total time: 2 seconds
lroot@localhost JSPBook1# _
```

图 A-10 执行 ant 指令

A-5 安装 Apache 2.0.48 + Tomcat 5.0.16

这里笔者要介绍如何在 Linux 下结合 Apache 2.0.48 和 Tomcat 5.0.16，目前笔者的环境如下：

- OS: Fedora Core 1
- J2SDK 1.4.2_03
- Apache 2.0.48
- Tomcat 5.0.16
- GNU Libtool 1.5
- Apache Portable Runtime (APR) 0.9.4

Apache 2 可以从 <http://httpd.apache.org/download.cgi> 网站自行免费下载，或者可以直接使用本书 CD 光盘中的 Apache 2.0.48，软件名称为：*httpd-2.0.48.tar.gz*。

第一步：安装 *httpd-2.0.48.tar.gz*；

下载完 *httpd-2.0.48.tar.gz* 后，将 *httpd-2.0.48.tar.gz* 解压缩，并执行下列指令：

- *tar zxvf httpd-2.0.48.tar.gz*

JSP2.0 技术手册

安装完后就执行 Apache 2。在 Linux 下执行 Apache 非常简单，只要执行 {Apache_Install}/bin 中的 apachectl 即可(例如：/usr/local/apach2/bin/apachectl)，指令如下：

- /usr/local/apache2/bin/apachectl start

注意

要关掉 apache2，则执行 /usr/local/apache2/bin/apachectl stop。

第三步：测试 Apache2：

打开 Netscape，输入 http://主机地址，假若 Apache 2 安装成功，则会看到如图 A-13 的画面：

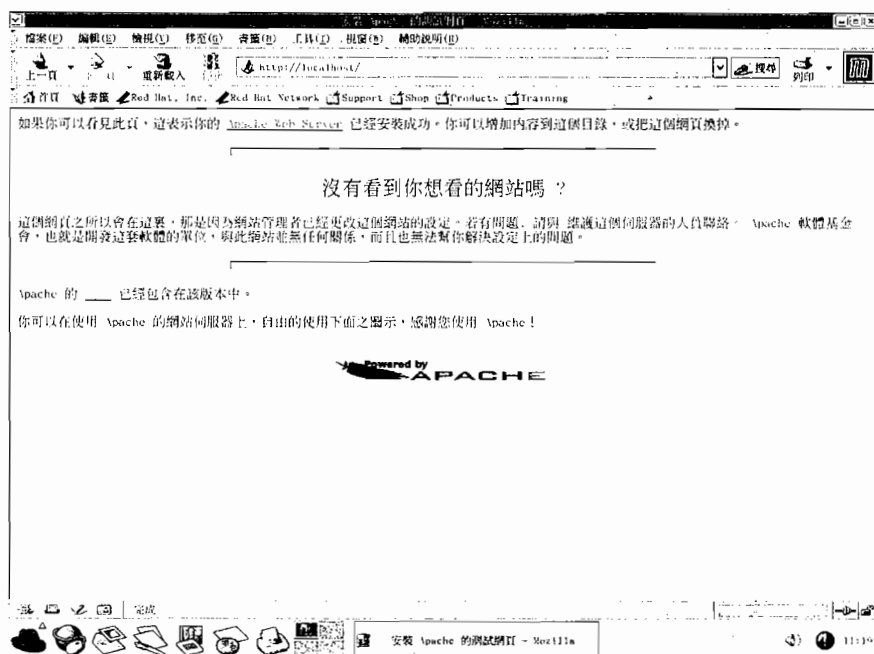


图 A-13 连接 http://主机地址/，测试 Apache 2.0.48

第四步：安装需要的套件：

Gnu Libtool 1.5 可以到 <http://www.gnu.org/order/ftp.html> 下载，笔者选择 <ftp://ftp.nctu.edu.tw/UNIX/gnu/> 免费下载，或者可以直接使用本书 CD 光盘中的 Gnu Libtool 1.5，软件名称为：
libtool-1.5.tar.gz。

下载完 *libtool-1.5.tar.gz* 后，将 *libtool-1.5.tar.gz* 解压缩并执行以下指令：

- tar zxvf libtool-1.5.tar.gz
- cd libtool-1.5
- ./configure

- make
- make install

Apache Portable Runtime (APR)可以到 <http://apr.apache.org> 选择站下载，笔者选择 <http://ftp.csie.nctu.edu.tw/data/apache/apr> 或者直接使用本书 CD 光盘中的 apr，软件名称为：*apr-0.9.4.tar.gz*。

下载完 *apr-0.9.4.tar.gz* 后，将 *apr-0.9.4.tar.gz* 解压缩并执行以下指令：

- tar zxvf apr-0.9.4.tar.gz
- cd apr-0.9.4
- ./configure
- make
- make install

第五步：安装 JK2；

JK2 可以到 <http://www.apache.org/dist/jakarta/tomcat-connectors/jk2/> 下载，或者可以直接使用本书 CD 光盘中的 jk2，软件名称为：*jakarta-tomcat-connectors-jk2-src-current.tar.gz*。

下载完 *jakarta-tomcat-connectors-jk2-src-current.tar.gz* 后，将它解压缩并执行以下指令：

- tar zxvf jakarta-tomcat-connectors-jk2-src-current.tar.gz
- cd jakarta-tomcat-connectors-jk2-2.0.2-src/jk/native2/
- ./configure --with-apxs2=/usr/local/apache2/bin/apxs \
- with-apache2-lib=/usr/local/apache2/modules/ \
- with-java-home=/usr/java/j2sdk1.4.2_03/ \
- with-java-platform=2 \
- enable-jni \
- with-apr-include=/usr/local/apr/include/apr-0/
- make

完成后可以在 *jakarta-tomcat-connectors-jk2-2.0.2-src/jk/build/jk2/apach2/* 下看到 *mod_jk2.so* 与 *jkjni.so*。

第六步：Apache2 + Tomcat5；

在完成上述五个步骤后，接下来就是要设定 Apache2 和 Tomcat5。首先将之前产生的 *mod_jk2.so* 复制到 apache 的 *modules* 目录下：

- cp -p mod_jk2.so /usr/local/apache2/modules/

编辑 */usr/local/apache2/conf/httpd.conf*，在 232 行附近 # *LoadModule foo_module modules/mod_foo.so* 下面一行加上 *LoadModule jk2_module modules/mod_jk2.so*，结果如图 A-14。

复制 *jakarta-tomcat-connectors-jk2-2.0.2-src/jk/conf/* 下的 *workers2.properties* 到

apache2/conf 下。

- `cp workers2.properties /usr/local/apache2/conf/`

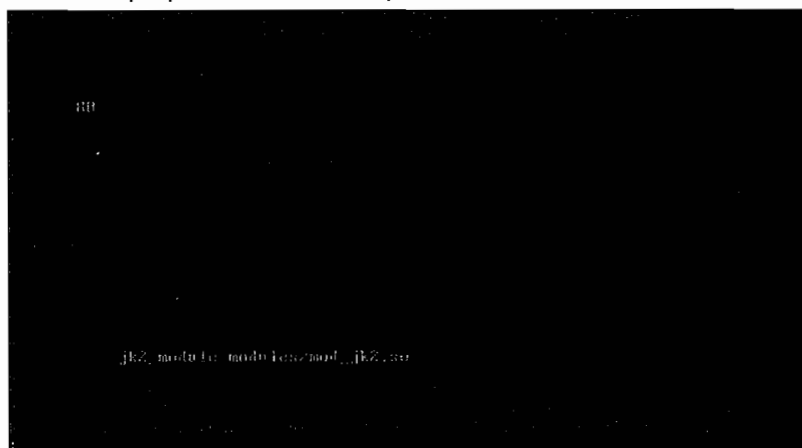


图 A-14 设定 httpd.conf

编辑 `worker2.properties` 如下：

■ `workers2.properties`

```
[shm:]
file=/usr/local/apache2/logs/jk2.shm
size=1000000
debug=0
disabled=0

[channel.socket:localhost:8009]
debug=0
tomcatId=localhost:8009

[uri:/jsp-examples]
```

注意

详细设定可参考 <http://jakarta.apache.org/tomcat/tomcat-4.1-doc/jk2/>，如果要新增自己的 webapp，可以通过 `[uri:/新增的 webapp 名称]` 来设定。

接下来到 `{Tomcat_Install}/conf/` 目录下编辑 `server.xml` (例如：`/usr/local/jakarta/jakarta-tomcat-5.0.16/conf/server.xml`)。将 8080 port 的 Connector mark 掉，并确定默认为 8009 port 的 AJP 1.3 Connector 没有被 mark 掉。

■ `server.xml`

```
<!-- Define a non-SSL Coyote HTTP/1.1 Connector on the port specified during
installation -->
<!-- ----- mark 起来
<Connector port="8080" maxThreads="150" minSpareThreads="25"
```

```

maxSpareThreads="75"
enableLookups="false" redirectPort="8443" acceptCount="100"
    debug="0" connectionTimeout="20000"
    disableUploadTimeout="true" />
--> <----- mark 起来
:
:
<!-- Define a Coyote/JK2 AJP 1.3 Connector on port 8009 -->
<Connector port="8009" enableLookups="false" redirectPort="8443"
debug="0"
    protocol="AJP/1.3" />

```

第七步：启动 Apache2 和 Tomcat5。

- `/usr/local/apache2/bin/apachectl start`
- `/usr/local/jakarta/jakarta-tomcat-5.0.16/bin/catalina.sh start`

打开 Netscape，输入 `http://主机地址/jsp-examples/`，假若设定成功，则会看到如图 A-15 的画面：

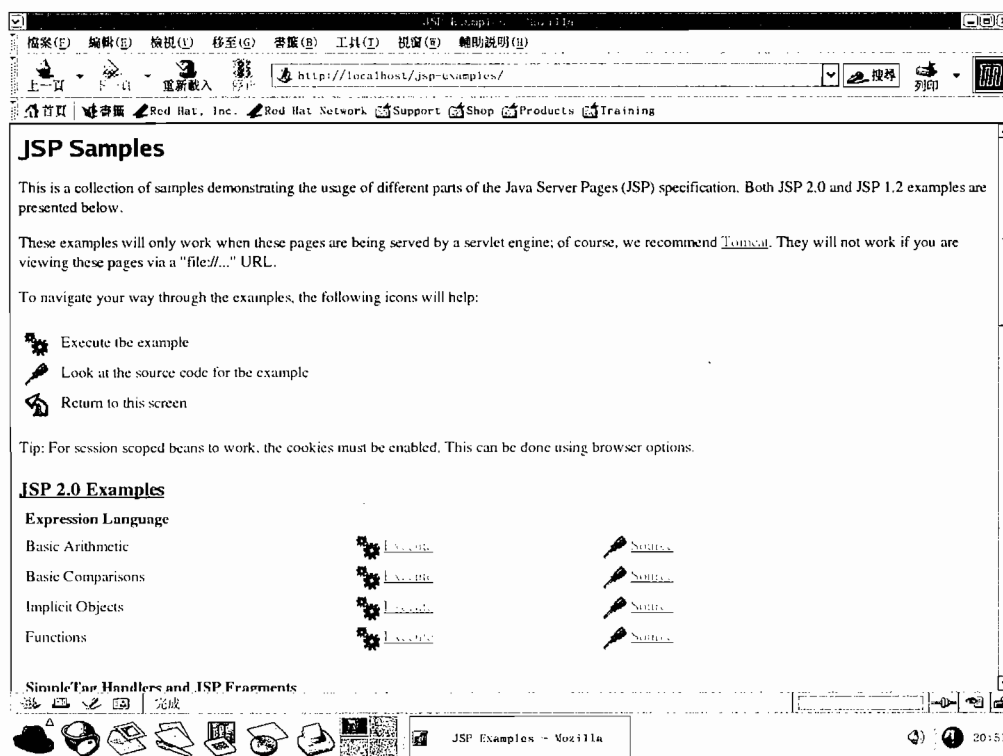


图 A-15 连接 `http://主机地址/jsp-examples/`，测试 Apache 2 和 Tomcat 5 的结合

JSP2.0 技术手册



附录 B

web.xml 元素介绍

每一个站的 *WEB-INF* 目录下都有一个 *web.xml* 的设定文件，它提供站台的配置设定。

web.xml 定义：

- 站台的名称和说明
- 针对环境参数(Context)做初始化的工作
- Servlet 的名称和对映
- Session 的设定
- Tag library 的对映
- JSP 网页设定
- Mime Type 的对映
- 错误处理
- 利用 JNDI 取得站台资源

要了解 *web.xml* 的设定值，必须先了解它的 schema，我们从 *web.xml* 中知道它的 schema 是由 Sun Microsystems 公司所制订的，有兴趣的读者可至 http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd 网页，那里有完整的解说。不过在此，我们把比较常用的设定为读者做个介绍。

以下是 Tomcat 中 example 的 *web.xml* 的设定：

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
  version="2.4">
.....
</web-app>
```

JSP2.0 技术手册

这是一般在写 XML 时所需要做的声明，定义了 xml 的版本、编码格式，还有最重要的指明 schema 的来源，为 http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd

<description>、<display-name>和<icon>

<description>站台的描述</description>

对站台做出描述。

<display-name>站台名称</display-name>

定义站台的名称。

<icon>

icon 元素包含 small-icon 和 large-icon 两个子元素。用来指定 web 站台中小图标和大图标的路径。

<small-icon>/路径/smallicon.gif</small-icon>

small-icon 元素应指向 web 站台中某个小图标的路径，大小为 16×16 pixel，但是图像文件必须为 GIF 或 JPEG 的格式，扩展名必须为：.gif 或.jpg。

<large-icon>/路径/largeicon.jpg</large-icon>

large-icon 元素应指向 web 站台中某个大图标的路径，大小为 32×32 pixel，但是图像文件必须为 GIF 或 JPEG 的格式，扩展名必须为：.gif 或.jpg。

</icon>

范例：

```
<display-name>JSPBook Examples</display-name>
<description>JSP 2.0 Tech Book's Examples</description>
<icon>
  <small-icon>/images/small.gif</small-icon>
  <large-icon>/images/large.gif</large-icon>
</icon>
<distributable>
```

<distributable>

distributable 元素为空标签，它的存在与否可以指定站台是否可分布式处理。如果 web.xml 中出现这个元素，则代表站台在开发时已经被设计为能在多个 JSP Container 之间分散执行。

范例：

```
<distributable/>
<context-param>
```

<context-param>

context-param 元素用来设定 web 站台的环境参数(Context)，它包含两个子元素：param-name 和 param-value。

<param-name>参数名称</param-name>

JSP2.0 技术手册

设定 Context 的名称。

```
<param-value>值</param-value>
```

设定 Context 名称的值

```
</context-param>
```

范例：

```
<context-param>
  <param-name>param_name</param-name>
  <param-value>param_value</param-value>
</context-param>
```

此所设定的参数，在 JSP 网页中可以使用下列方法来取得：

```
${initParam.param_name}
```

若在 Servlet 中可以使用下列方法来取得：

```
String param_value = getServletContext().getInitParameter("param_name");
```

```
<filter>
```

```
<filter>
```

filter 元素用来声明 filter 的相关设定。filter 元素除了下面将介绍的子元素之外，还包括 <servlet>介绍过的<icon>、<display-name>、<description>、<init-param>，其用途一样。

```
<filter-name>Filter 的名称</filter-name>
```

定义 Filter 的名称。

```
<filter-class>Filter 的类名称</filter-class>
```

定义 Filter 的类名称。例如：com.foo.hello

```
</filter>
```

范例：

```
<filter>
  <filter-name>setCharacterEncoding</filter-name>
  <filter-class>tw.com.javaworld.CH11.SetCharacterEncodingFilter</filter-class>
  <init-param>
    <param-name>encoding</param-name>
    <param-value>GB2312</param-value>
  </init-param>
</filter>
```

```
<filter-mapping>
```

```
<filter-mapping>
```

filter-mapping 元素的两个主要子元素 filter-name 和 url-pattern。用来定义 Filter 所对应的 URL。

```
<filter-name>Filter 的名称</filter-name>
```

定义 Filter 的名称。

`<url-pattern>URL</url-pattern>`

Filter 所对应的 URL。例如：`<url-pattern>/Filter/Hello</url-pattern>`

`<servlet-name>` Servlet 的名称`</servlet-name>`

定义 Servlet 的名称。

`<dispatcher>REQUEST | INCLUDE | FORWARD | ERROR</dispatcher>`

设定 Filter 对应的请求方式，有 REQUEST、INCLUDE、FORWARD、ERROR 四种，默认为 REQUEST。

`</filter-mapping>`

范例：

```
<filter-mapping>
  <filter-name>GZIPEncoder</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

`<listener>`

`<listener>`

listener 元素用来设定 Listener 接口，它的主要子元素为`<listener-class>`。

`<listener-class>`Listener 的类名称`</listener-class>`

定义 Listener 的类名称。例如：com.foo.hello

`</listener>`

范例：

```
<listener>
  <listener-class>tw.com.javaworld.CH11.ContextListener</listener-class>
</listener>
```

`<servlet>`

`<servlet>`

servlet 元素声明一个 servlet 的数据，如果有 jsp-file 元素和 load-on-startup 元素存在，则 JSP 网页就会被重新编译和加载。servlet 元素除了下面将介绍的子元素之外，还包括上述介绍过的`<icon>`、`<display-name>`、`<description>`，其用途一样。

`<servlet-name>`Servlet 的名称`</servlet-name>`

定义 Servlet 的名称。

`<servlet-class>`Servlet 的类名称`</servlet-class>`

定义 Servlet 的类名称。例如：com.foo.hello

`<jsp-file>/路径/xxx.jsp</jsp-file>`

在 web 站台中，某一个 JSP 网页的完整路径。

`<init-param>`

init-param 元素包含三个子元素, 分别为 param-name、param-value、description, 用来初始化参数。在 Servlet 中, 可以在 init()方法中使用 ServletConfig 对象的 getInitParameter()方法来取得初始化参数值:

```
String value = ServletConfig().getInitParameter("name");
```

```
<param-name>参数名称</param-name>
```

定义参数名称。

```
<param-value>参数的值</param-value>
```

定义指定参数的值。

```
<description>参数的描述</description>
```

参数的描述。

```
</init-param>
```

```
<load-on-startup>数字</load-on-startup>
```

load-on-startup 元素表示 web 站台被启动时, 此 Servlet 会被自动加载执行。此元素的内容必须为一正整数, 用来自定义这个 Servlet 被加载的优先级。若设为 1, 则表示最早被加载执行, 然后依此类推。

```
</servlet>
```

范例:

```
<servlet>
  <servlet-name>ServletConfigurator</servlet-name>
  <servlet-class>
    org.logicalcobwebs.proxool.configuration.ServletConfigurator
  </servlet-class>

  <init-param>
    <param-name>propertyFile</param-name>
    <param-value>WEB-INF/classes/Proxool.properties</param-value>
  </init-param>

  <load-on-startup>1</load-on-startup>
</servlet>
```

```
<servlet-mapping>
```

```
<servlet-mapping>
```

servlet-mapping 元素包含两个子元素 servlet-name 和 url-pattern。用来定义 Servlet 所对应的 URL。

```
<servlet-name>Servlet 的名称</servlet-name>
```

定义 Servlet 的名称。

```
<url-pattern>URL</url-pattern>
```

Servlet 所对应的 URL。例如: <url-pattern>/Servlet/Hello</url-pattern>

```
</servlet-mapping>
```


范例：

```
<servlet-mapping>
  <servlet-name>LoginChecker</servlet-name>
  <url-pattern>/LoginChecker</url-pattern>
</servlet-mapping>
```

```
<session-config>
```

```
<session-config>
```

session-config 包含一个子元素 session-timeout。定义 web 站台中的 session 参数。

```
<session-timeout>分钟</session-timeout>
```

定义这个 web 站台所有 session 的有效期限。单位为分钟。

```
</session-config>
```

范例：

```
<session-config>
  <session-timeout>90</session-timeout>
</session-config>
```

```
<mime-mapping>
```

```
<mime-mapping>
```

mime-mapping 包含两个子元素 extension 和 mime-type。定义某一个扩展名和某一 MIME Type 做对映

```
<extension>扩展名名称</extension>
```

扩展名名称。

```
<mime-type>MIME 格式</mime-type>
```

MIME 格式。

```
</mime-mapping>
```

范例：

```
<mime-mapping>
  <extension>doc</extension>
  <mime-type>application/vnd.ms-word</mime-type>
</mime-mapping>
<mime-mapping>
  <extension>xls</extension>
  <mime-type>application/vnd.ms-excel</mime-type>
</mime-mapping>
<mime-mapping>
  <extension>ppt</extension>
  <mime-type>application/vnd.ms-powerpoint</mime-type>
</mime-mapping>
```

```
<welcome-file-list>
```

```
<welcome-file-list>
```

welcome-file-list 包含一个子元素 welcome-file。用来定义首页的列单。

JSP2.0 技术手册

`<welcome-file>`指定首页的文件名称`</welcome-file>`

`welcome-file` 用来指定首页的文件名称。我们可以用`< welcome-file >`指定好几个首页，而服务器会依设定的顺序来寻找首页，假设找不到第一个`< welcome-file >`所指定的文件，就会依序找第二个，以此类推。

范例：

```
<welcome-file-list>
  <welcome-file>index.jsp</welcome-file>
  <welcome-file>index.html</welcome-file>
</welcome-file-list>
```

`<error-page>`

`<error-page>`

`error-page` 元素包含三个子元素 `error-code`、`exception-type` 和 `location`。将错误代码(Error Code)或异常(Exception)的种类对应到 web 站台的资源路径。

`<error-code>`错误代码`</error-code>`

HTTP Error Code，例如：404

`<exception-type>`Exception`</exception-type>`

一个完整类名称的 Java 异常类型。

`<location>`/路径`</location>`

在 web 站台内的相关资源路径。

`</error-page>`

范例：

```
<error-page>
  <error-code>404</error-code>
  <location>/error404.jsp</location>
</error-page>
<error-page>
  <exception-type>java.lang.Exception</exception-type>
  <location>/except.jsp</location>
</error-page>
```

`<jsp-config>`

`<jsp-config>`

`<jsp-config>` 元素主要用来设定 JSP 的相关配置，`<jsp-config>` 包括 `<taglib>` 和 `<jsp-property-group>` 两个子元素。其中 `<taglib>` 元素在 JSP 1.2 时就已经存在；而 `<jsp-property-group>` 是 JSP 2.0 新增的元素。

`<taglib>`

`taglib` 元素包含两个子元素 `taglib-uri` 和 `taglib-location`。用来设定 JSP 网页所用到的 Tag Library 路径。

```
<taglib-uri>URI</taglib-uri>
```

taglib-uri 定义 TLD 文件的 URI, JSP 网页的 taglib 指令可以经由这个 URI 存取到 TLD 文件。

```
<taglib-location>/WEB-INF/lib/mytag.jar</taglib-location>
```

TLD 文件相对 Web 站台的存放位置。

```
</taglib>
```

```
<jsp-property-group>
```

jsp-property-group 元素主要有八个元素, 分别为:

```
<description>Description</description>
```

此设定的说明。

```
<display-name>Name</display-name>
```

此设定名称。

```
<url-pattern>URL</url-pattern>
```

设定值所影响的范围, 如: /CH2 或 /*.jsp

```
<el-ignored>true | false</el-ignored>
```

若为 true, 表示不支持 EL 语法。

```
<scripting-invalid>true | false</scripting-invalid>
```

若为 true, 表示不支持<% scripting %>语法

```
<page-encoding>encoding</page-encoding>
```

设定 JSP 网页的编码

```
<include-prelude>.jspf</include-prelude>
```

设置 JSP 网页的抬头, 扩展名为.jspf

```
<include-coda>.jspf</include-coda>
```

设置 JSP 网页的结尾, 扩展名为.jspf

```
</jsp-property-group>
```

```
</jsp-config>
```

范例:

```
<jsp-config>
<taglib>
  <taglib-uri>Taglib</taglib-uri>
  <taglib-location>/WEB-INF/tlds/MyTaglib.tld</taglib-location>
</taglib>

  <jsp-property-group>
    <description>
      Special property group for JSP Configuration JSP example.
    </description>
```

JSP2.0 技术手册

```

        <display-name>JSPConfiguration</display-name>
        <url-pattern>/jsp/* </url-pattern>
        <el-ignored>true</el-ignored>
        <page-encoding>GB2312</page-encoding>
        <scripting-invalid>true</scripting-invalid>
        <include-prelude>/include/prelude.jspf</include-prelude>
        <include-coda>/include/coda.jspf</include-coda>
    </jsp-property-group>
</jsp-config>
<resource-ref>

```

```
<resource-ref>
```

resource-ref 元素包含五个子元素 description、res-ref-name、res-type、res-auth 和 res-sharing-scope。利用 JNDI 取得站台可利用的资源。

```
<description>说明</description>
```

资源说明

```
<res-ref-name>资源名称</res-ref-name>
```

资源名称

```
<res-type>资源种类</res-type>
```

资源种类

```
<res-auth>Application | Container</res-auth>
```

资源经由 Application 或 Container 来许可

```
<res-sharing-scope>Shareable | Unshareable</res-sharing-scope>
```

资源是否可以共享。默认值为 Shareable

```
</resource-ref>
```

范例：

```

<resource-ref>
    <description>JNDI JDBC DataSource of JSPBook</description>
    <res-ref-name>jdbc/sample_db</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
</resource-ref>

```

结论：

以上把比较常用的设定提来说明，若读者需要其他更详尽的设定规范，请直接到 http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd，相信你会有更多的收获。



附录 C

使用 JDBC-ODBC 桥接器连接 Access

一般来说:我们所使用的**测试环境**通常都是在 Type 1 的 JDBC 驱动程序底下,再通过 ODBC 去连接数据库。因为 JDK 里面已经内附 Sun 的 JDBC-ODBC 桥接器,在你所撰写的 Java 应用程序,不需要特别去构建 JDBC-ODBC 桥接器的环境,但是 JDBC 还要通过 ODBC 这一层才能存取到数据库,所以我们需要构建 ODBC 到数据库的环境设定。

首先我们建立数据库,在这里以 Microsoft Access 为基本的数据库系统建立一个名叫 sample 的数据库,数据库文件名: *sample.mdb*; 表格名称: employee, 如下:

字段名称	数据类型
Employ_id	自动编号
Name	文字
Old	数字

接下来设定 ODBC 数据源,你必须遵照下列步骤进行:

1. 从 Windows 控制台中数据源(ODBC)选项来打开 ODBC 数据源管理器。若是 Windows 2000 的读者, ODBC 数据源从【开始】→【设置】→【控制面板】→【系统】→【管理工具】→【数据源 (ODBC)】

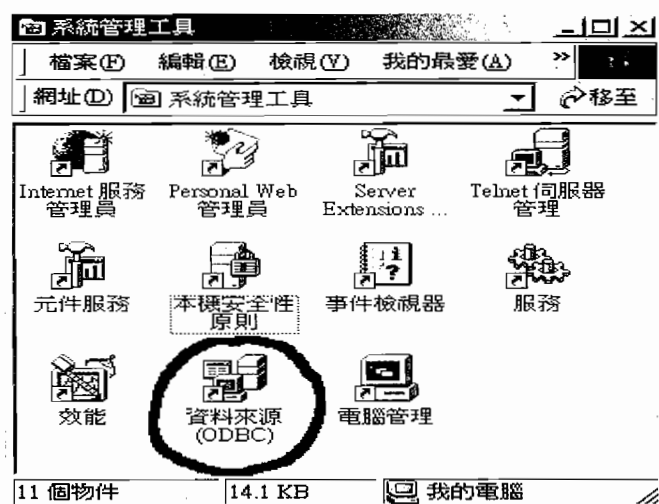


图 C-1 数据源 ODBC 的路径位置

打开【数据源 (ODBC)】选项之后，读者可看到如图 C-2 的对话框，并且选【系统 DNS】。

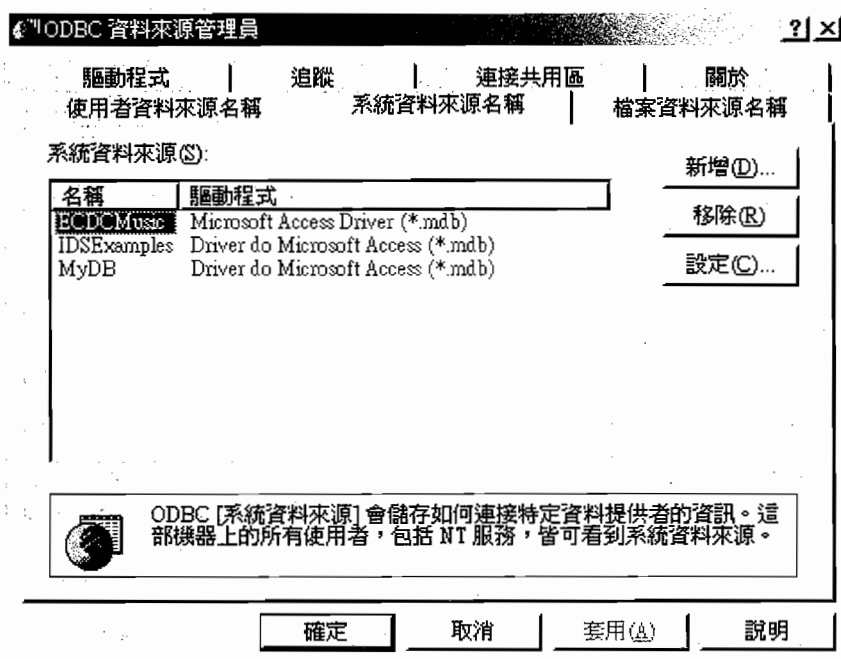


图 C-2 数据源 (ODBC) | 系统 DNS

2. 选择【添加】按钮将【创建新的数据源】加入，如图 C-3：



图 C-3 创建新的数据源

3. 选择【Microsoft Access Driver(*.mdb)】并且按【完成】。你就会看到 ODBC Microsoft Access 安装，如图 C-4：

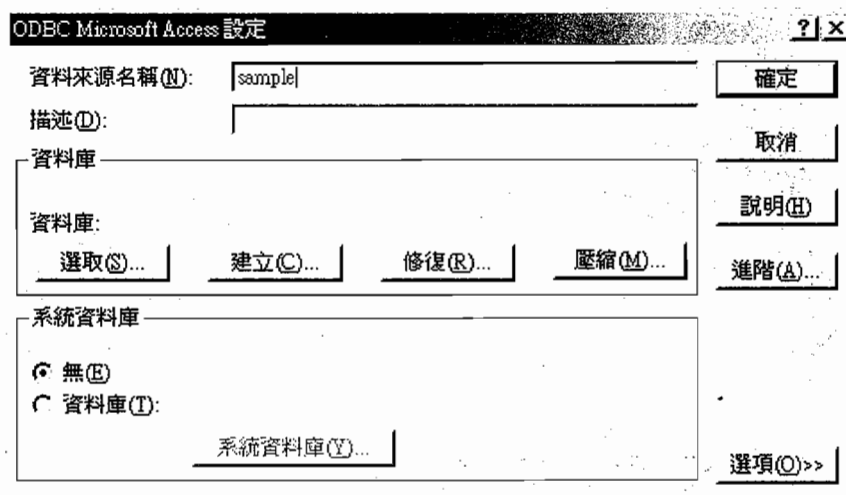


图 C-4 ODBC Microsoft Access 安装

4. 在数据源名填入 sample 并在数据库栏按【选择】，就会出现选择数据库画面，如图 C-5：

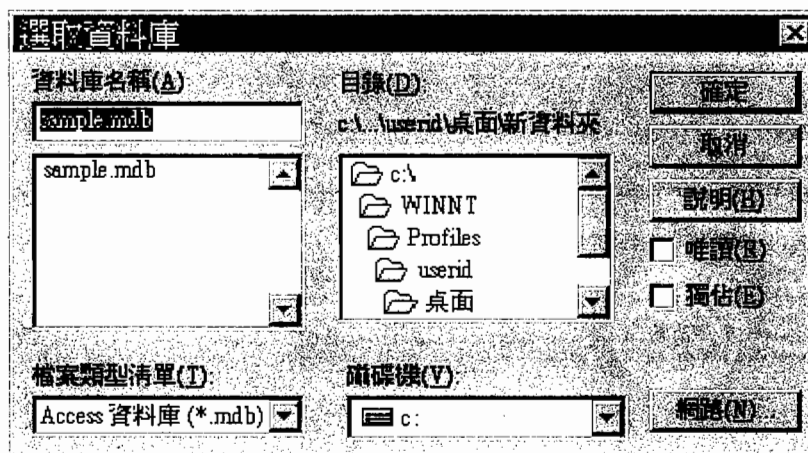


图 C-5 选择数据库

5. 寻找你放置 sample.mdb 的路径，并且选它，再按【确定】钮。你就会看到 ODBC Microsoft Access 安装已经改变，按【确定】来同意你的改变。

6. 此时你将在 ODBC 数据源管理器的【用户 DNS】列表内，看到 sample 数据源在这个列表上。按下【确定】来关闭窗口。



附录 D

JSP 资源

■ 国外资源

- Sun 官方 JavaServlet 技术首页
<http://java.sun.com/products/servlet/index.jsp>
- Sun 官方 JavaServer Pages 技术首页
<http://java.sun.com/products/jsp/index.jsp>
- Sun 官方 JavaServer Pages 讨论社群
<http://forum.java.sun.com/forum.jsp?forum=45>
- Sun 官方 Java Servlet 讨论社群
<http://forum.java.sun.com/forum.jsp?forum=33>
- Sun 官方 JDBC 讨论社群
<http://forum.java.sun.com/forum.jsp?forum=48>
- TheServerSide - Servlets, JSP, JSTL, and Frameworks 讨论区
http://www.theserverside.com/discussions/forum.jsp?forum_id=4

■ 繁体资源

- 台湾 JavaWorld Java 技术论坛
<http://www.javaworld.com.tw/jute> 或 <http://www.jsptw.com/jute>
- JSP 学习讲义
<http://www.jsp.mlc.edu.tw/>

JSP2.0 技术手册



附录 E

HTTP 状态码

表 E-1 列出了由 HttpServletResponse 接口定义的 HTTP 状态码常数，可用于 setStatus() 和 sendError() 的参数。Servlet 2.0 API 新增 HTTP Version 1.1 的状态码。HTTP 1.1 的状态码需要支持 HTTP 1.1 的浏览器才能使用，目前的 IE 4.0 以上版本皆已支持 HTTP 1.1。

表 E-1 HTTP 状态码常数

常 数	码	默认讯息	意 义	HTTP 版本
SC_CONTINUE	100	Continue	Server 已收到由 client 端传来的部分请求，并可继续传递请求剩下的部分	1.1
SC_SWITCHING_PROTOCOLS	101	Switching Protocols	Server 同意依照 client 请求中的 Upgrade 标头，改用其他协议。(包括更新的 HTTP 版本)	1.1
SC_OK	200	OK	Client 请求成功	1.0
SC_CREATED	201	Created	Server 产生一个资源(resource)，可能是为了响应 client 的请求。响应本体 (response body)应包含指向该新资源的 URL。若 server 无法立即产生该份资源，应改用 SC_ACCEPTED	1.0
SC_ACCEPTED	202	Accepted	请求已被接受，但是 server 还没处理完 client 端的请求	1.0
SC_NON_AUTHORITATIVE_INFORMATION	203	Non-Authoritative Information	HTTP 响应标头 (response header)源自其他来源，非原 server 所发出。一般 servlet 没有理由使用这个状态码	1.1

JSP2.0 技术手册

续表

常 数	码	默认讯息	意 义	HTTP 版本
SC_NO_CONTENT	204	No Content	请求成功，但不传回新的响应本体 (response body)。浏览器收到此状态码时，应保持原来的显示内容	1.0
SC_RESET_CONTENT	205		要求浏览器应重新加载目前所显示的文件。本状态码适合用于想让浏览器显示更新过的状态	1.1
SC_PARTIAL_CONTENT	206	Partial Content	Server 已处理好部分的 GET 请求，并传回 client 用 Range 标头指定的部分数据	1.1
SC_MULTIPLE_CHOICES	300	Multiple Choices	请求的 URL 指向一个以上的资源，如一个 URL 可能会指向多个不同语言版本的文件。响应本体 (response body) 应以合适的内容类型 (content type) 向 client 解释选项。Server 可用 Location 标头向 client 建议选用哪一个	1.1
SC_MOVED_PERMANENTLY	301	Moved Permanently	所请求的资源已永久搬至新地址，将来要参考它均应使用新地址。新地址由 Location 标头指定。大部分的浏览器收到此标头后，都会自动前往新地址	1.0
SC_MOVED_TEMPORARILY	302	Moved Temporarily	所请求的资源暂时移至新地址，但将来要参考它应使用旧地址。新地址由 Location 标头指定。几乎所有浏览器收到此标头后都会自动前往新地址	1.0
SC_SEE_OTHER	303	See Other	已处理所请求的资源，但 Client 应往 Location 标头所指定的 URL 以 GET 取得响应内容	1.1
SC_NOT_MODIFIED	304	Not Modified	所请求的文件自从 If-Modified-Since 标头所说的时间到现在都还没更新。这个状态码较少被我们使用，通常我们都是实现 getLastModified()	1.0
SC_USE_PROXY	305	Use Proxy	所请求的资源应向 Location 标头指定的 Proxy Server 取得	1.1
SC_BAD_REQUEST	400	Bad Request	Server 无法了解请求的内容，可能是语法错误	1.0
SC_UNAUTHORIZED	401	Unauthorized	请求未经授权 (authorization)。与 WWW-Authenticate 和 Authorization 标头并用	1.0

JSP2.0 技术手册

续表

常 数	码	默认讯息	意 义	HTTP 版本
SC_FORBIDDEN	403	Forbidden	Server 收到请求, 但不想提供服务。Server 可在响应本体 (response body) 解释不想服务的原因	1.0
SC_NOT_FOUND	404	Not Found	找不到所请求的资源, 或该资源无法使用。这个状态码一般也是常见之一	1.0
SC_METHOD_NOT_ALLOWED	405	Method Not Allowed	此 URL 不支持 Client 所请求的使用方法 (request method, 如 GET、POST)	1.1
SC_NOT_ACCEPTABLE	406	Not Acceptable	所请求的资源是存在的, 但是不被 Client 所接受的类型, 因此不提供服务	1.1
SC_PROXY_AUTHENTICATION_REQUIRED	407	Proxy Authentication Required	Proxy Server 做进一步处理前需进行授权。与 Proxy-Authenticate 标头并用	1.1
SC_REQUEST_TIMEOUT	408	Request Timeout Client	位于 Server 设定的一段时间内送达整个请求所产生的状态码	1.1
SC_CONFLICT	409	Conflict	无法完成请求, 因为它与其他请求或 Server 的设定发生冲突	1.0
SC_GONE	410	Gone	Server 没有请求的资源, 且不知道该用哪个地址才能找到它。资源已永久移除时, 才可使用此状态码	1.1
SC_LENGTH_REQUIRED	411	Length Required	没有 Content-Length 标头, Server 就不接受请求	1.1
SC_PRECONDITION_FAILED	412	Precondition Failed	请求中形如 if... 的标头所指定的“前提条件” (precondition) 执行为假 (false)	1.1
SC_REQUEST_ENTITY_TOO_LARGE	413	Request Entity Too Large	Server 不处理请求, 因为所请求的内容太大。若此限制是暂时性的, Server 可用 Retry-After 标头告知 client 过多久再试试	1.1
SC_REQUEST_URI_TOO_LONG	414	Request URI Too Long	Server 不处理请求, 因为请求所使用的 URI 超过 Server 的限制。Client 不小心把 POST 请求当 GET 请求使用时, 就会发生这种情况	1.1
SC_UNSUPPORTED_MEDIA_TYPE	415	Unsupported Media Type	Server 不处理请求, 因为不支持请求内容的类型	1.1

JSP2.0 技术手册

续表

常 数	码	默认讯息	意 义	HTTP 版本
SC_INTERNAL_SERVER_ERROR	500	Internal Server Error	内部发生不可预期的 Server 错误，故无法完成 Client 的请求。这个状态码也是在写 Servlet 或 JSP 时常见到的状态码	1.0
SC_NOT_IMPLEMENTED	501	Not Implemented	Server 不支持完成 Client 请求所需的功能	1.0
SC_BAD_GATEWAY	502	Bad Gateway	身为 gateway 或 proxy 的 server 没有由上游 server 收到有效的响应	1.0
SC_SERVICE_UNAVAILABLE	503	Service Unavailable	服务 (Server)暂时无法使用，但将来会恢复	1.0
SC_GATEWAY_TIMEOUT	504	Gateway Timeout	身为 gateway 或 proxy 的 server 在默认的一段时间内没有由上游 server 收到有效的响应	1.1
SC_HTTP_VERSION_NOT_SUPPORTED	505	HTTP Version Not Supported	Server 不支持请求使用的 HTTP 协议的版本	1.1



附录 F

ASCII 码

十进位	十六进位	符号	十进位	十六进位	符号	十进位	十六进位	符号
0	0x00	NUL	21	0x15	NAK	42	0x2A	*
1	0x01	SOH	22	0x16	SYN	43	0x2B	+
2	0x02	STX	23	0x17	ETB	44	0x2C	,
3	0x03	ETX	24	0x18	CAN	45	0x2D	-
4	0x04	EOT	25	0x19	EM	46	0x2E	.
5	0x05	ENQ	26	0x1A	SUB	47	0x2F	/
6	0x06	ACK	27	0x1B	ESC	48	0x30	0
7	0x07	BEL	28	0x1C	FS	49	0x31	1
8	0x08	BS	29	0x1D	GS	50	0x32	2
9	0x09	HT	30	0x1E	RS	51	0x33	3
10	0x0A	LF	31	0x1F	US	52	0x34	4
11	0x0B	VT	32	0x20	SP	53	0x35	5
12	0x0C	FF	33	0x21	!	54	0x36	6
13	0x0D	CR	34	0x22	"	55	0x37	7
14	0x0E	SO	35	0x23	#	56	0x38	8
15	0x0F	SI	36	0x24	\$	57	0x39	9
16	0x10	DLE	37	0x25	%	58	0x3A	:
17	0x11	DC1	38	0x26	&	59	0x3B	;
18	0x12	DC2	39	0x27	'	60	0x3C	<
19	0x13	DC3	40	0x28	(61	0x3D	=
20	0x14	DC4	41	0x29)	62	0x3E	>

JSP2.0 技术手册



附录 G

Apache License 1.1

/*

```
=====
* The Apache Software License, Version 1.1
*
* Copyright (c) 2000 The Apache Software Foundation. All rights
* reserved.
*
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions
* are met:
*
* 1. Redistributions of source code must retain the above copyright
* notice, this list of conditions and the following disclaimer.
*
* 2. Redistributions in binary form must reproduce the above copyright
* notice, this list of conditions and the following disclaimer in
* the documentation and/or other materials provided with the
* distribution.
*
* 3. The end-user documentation included with the redistribution,
* if any, must include the following acknowledgment:
* "This product includes software developed by the
* Apache Software Foundation (http://www.apache.org/)."
```

Alternately, this acknowledgment may appear in the software itself,
if and wherever such third-party acknowledgments normally appear.

JSP2.0 技术手册

* 4. The names "Apache" and "Apache Software Foundation" must
* not be used to endorse or promote products derived from this
* software without prior written permission. For written
* permission, please contact apache@apache.org.
*

* 5. Products derived from this software may not be called "Apache",
* nor may "Apache" appear in their name, without prior written
* permission of the Apache Software Foundation.
*

* THIS SOFTWARE IS PROVIDED "AS IS" AND ANY EXPRESSED OR IMPLIED
* WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
* OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
* DISCLAIMED. IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR
* ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
* SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
* LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF
* USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
* ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
* OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
* OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
* SUCH DAMAGE.
*

=====

*
* This software consists of voluntary contributions made by many
* individuals on behalf of the Apache Software Foundation. For more
* information on the Apache Software Foundation, please see
* <http://www.apache.org/>.
*

* Portions of this software are based upon public domain software
* originally written at the National Center for Supercomputing Applications,
* University of Illinois, Urbana-Champaign.
*/

《JSP2.0 技术手册》读者调查表

尊敬的读者：

感谢您对我们的支持与爱护。为了今后为您提供更优秀的图书，请您抽出宝贵的时间将您的意见以下表的方式及时告知我们（可另附页）。我们将从中评选出热心读者若干名，免费赠阅我们以后出版的图书。

您的意见是我们创造精品的动力源泉！

姓名：_____ 性别：☐ 男 ☐ 女 年龄：_____ 职业：_____
电话（寻呼）：_____ E-mail：_____
传真：_____ 通信地址：_____
邮编：_____

1. 影响您购买本书的因素（可多选）：

☐封面封底 ☐价格 ☐内容提要、前言和目录 ☐书评广告 ☐出版物名声
☐作者名声 ☐正文内容 ☐其他_____

2. 您对本书的满意度：

从技术角度 ☐很满意 ☐比较满意 ☐一般 ☐较不满意 ☐不满意
☐改进意见_____

从文字角度 ☐很满意 ☐比较满意 ☐一般 ☐较不满意 ☐不满意
☐改进意见_____

从版面、封面设计角度 ☐很满意 ☐比较满意 ☐一般 ☐较不满意
☐不满意 ☐改进意见_____

3. 您最喜欢书中的哪篇（或章、节）？请说明理由。

4. 您最不喜欢书中的哪篇（或章、节）？请说明理由。

5. 您希望本书在哪些方面进行改进？

6. 您感兴趣或希望增加的图书选题有：

请寄：武汉市洪山区吴家湾紫菘花园 16 栋西门 401 周筠 收（430074）

E-mail: yeka@csdn.net

yeka@broadview.com.cn

电话：027-87691935

博文视点资讯有限公司 (BROADVIEW Information Co.,Ltd.) 是信息产业部直属的中央一级科技与教育出版社——电子工业出版社 (PHEI) 与国内最大的 IT 技术网站 CSDN.NET 和最具专业水准的 IT 杂志社《程序员》合资成立的以 IT 图书出版为主业、开展相关信息和知识增值服务的资讯公司。

我们的理念是：创新专业出版体制；培养职业出版队伍；打造精品出版品牌；完善全面出版服务。

秉承博文视点的理念，博文视点的产品线为面向 IT 专业人员的出版物和相关服务。博文视点将重点做好以下工作：

- (1) 在技术领域开发专业作（译）者群体和高质量的原创图书
- (2) 在图书领域建立专业的选题策划和审读机制
- (3) 在市场领域开创有效的宣传手段和营销渠道

博文视点有效地综合了电子工业出版社、《程序员》杂志社和 CSDN.NET 的资源和人才，建立全新专业的立体出版机制，确立独特的出版特色和优势，将打造 IT 出版领域的著名品牌，并力争成为中国最具影响力的专业 IT 出版和服务提供商。

作为合资公司，博文视点的团队融合了各方面的精英力量：原电子工业出版社 IT 图书专业出版实力的代表部门——计算机图书事业部的团队；《程序员》杂志社和 CSDN 网站的主创人员；著名 IT 专业图书策划人周筠女士及其创作群。这是一个整合专业技术人员和专业出版人员的团队；这是一个充满创新意识和创作激情的团队；这是一个不断进取、追求卓越的团队。

电子工业出版社与《程序员》杂志和 CSDN 网站的合作以最有效率的方式形成了出版资源、媒体资源、网络资源的整合和互动，成为 2003 年 IT 出版界备受瞩目的事件。

“技术凝聚实力，专业创新出版”，BROADVIEW 与您携手共迎信息时代的机遇与挑战！



博文视点

地址：北京市复兴路 47 号天行建商务大厦 604 室

邮 编：100036

总 机：010-51922832 传 真：010-51922823

作者读者服务部（图书）：010-51922839 国外作者写作、引进版图书：010-51922825

http://www.broadview.com.cn 投稿及读者反馈：editor@broadview.com.cn

分馆地址：武汉市洪山区吴家湾紫菀花园 16 栋西门 401 邮编：430074

电话：027-87691935 E-mail:yeka@csdn.net